INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

University of Alberta

Type System for an Object-Oriented Database Programming Language

by

Yuri Leontiev ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

Department of Computing Science

Edmonton, Alberta
Fall 1999

National Library of Canada

Acquisitions and Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada

Acquisitions et services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-46872-0

Canada

University of Alberta

Library Release Form

Name of Author: Yuri Leontiev

Title of Thesis: Type System for an Object-Oriented Database Programming Language

Degree: Doctor of Philosophy

Year this Degree Granted: 1999

Permission is hereby granted to the University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves all other publication and other rights in association with the copyright in the thesis, and except as hereinbefore provided, neither the thesis nor any substantial portion thereof may be printed or otherwise reproduced in any material form whatever without the author's prior written permission.

[illegible]

Yuri Leontiev
10710 83 Ave #204
Edmonton, AB
Canada, T6E 2E4

Date: 4 October 1999

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Type System for an Object-Oriented Database Programming Language** submitted by Yuri Leontiev in partial fulfillment of the requirements for the degree of **Doctor of Philosophy**.

[illegible]

Dr. M. T. Özsu

[illegible]

Dr. D. S. Szafron

[illegible]

Dr. M. P. Atkinson

[illegible]

Dr. W. W. Armstrong

[illegible]

Dr. L. W. Pedrycz

Date: 4 October 1999

To my parents Vladimir Vasilievitch Leontiev and Alla Davidovna Leontieva

Abstract

The concept of an object-oriented database programming language (OODBPL) is appealing because it has the potential of combining the advantages of object orientation and database programming to yield a powerful and universal programming language design.

A uniform and consistent combination of object orientation and database programming, however, is not straightforward. Since one of the main components of an object-oriented programming language is its type system, one of the first problems that arise during an OODBPL design is related to the development of a uniform, consistent, and theoretically sound type system that is sufficiently expressive to satisfy the combined needs of object orientation and database programming.

This dissertation presents the design of a type system suitable for object-oriented database programming. The type system has a unique combination of uniformity, expressibility, verifiability, and theoretically proven soundness. It also possesses features that make it suitable for database programming, such as seamless integration of imperative types and features, precise query typing via union and intersection types, separation among three abstraction layers providing a high degree of code reuse, parametric polymorphism, extensibility, and dynamic type analysis capabilities.

In the process of type system development, a theoretical framework for dealing with type systems that combine parametric and inclusion polymorphism is established. Due to its modular construction, this framework can be easily extended and used beyond the scope of this dissertation. Another contribution of this work is an extensive analysis of existing and proposed type systems from the point of view of the set of requirements related to object orientation and database programming.

This research leads to the development of a uniform and theoretically sound OODBPL that can successfully utilize the power inherent in both object orientation and database programming paradigms. This will eventually lead to the development and implementation of a uniform object-oriented database system that will use the OODBPL as its main programming and query engine.

Acknowledgements

I would like to express my sincere thanks to my supervisors Dr. Tamer Özsu and Dr. Duane Szafron for their invaluable input, constant support and understanding. This dissertation would not have been possible without their guidance.

I am also grateful to the members of my committee Dr. Malcolm Atkinson, Dr. William Armstrong, and Dr. Witold Pedrycz for many valuable comments and suggestions.

Special thanks go to my Russian supervisor Vladimir Lvovitch Arlazarov and my colleagues Alexander Merkov and Marina Iofinova. Without their guidance and support this work would not have been possible, as I would not have had an opportunity to come to study at the University of Alberta. I would also like to thank Dr. Tony Marsland who helped me come to this University.

Thanks also go to the database research group and in particular to Boman Irani, Iqbal Goralwalla, Kaladhar Voruganti, Paul Iglinski, and Wade Holst for many interesting, helpful, and insightful discussions.

Thanks to the University of Alberta, Department of Computing Science, and the Killam Fund for their financial support.

My supervisors at Intuit Canada Ltd. Andrew Smith and Drew LaHaie have provided me with an opportunity to continue my studies while working full-time for this company.

Many thanks to my friends Igor Kolodkin, Igor Voronkov, Oleg Verevka and Victoria Lohvin, Alexandr and Alina Litvak, Andrey Zelnikov, Ilya Vodopyanov, Andrey Maximov, Vlad Alexiev and Dolya Alexieva, Anna Zolotova, Ewgenii Gawrilow and Tatyana Gawrilowa, Grigoriy Andronov, Dmitriy Makarov and Natalya Makarova, Maxim Potashev, Maria Schigoleva, Dmitriy Khabirov and Olga Chernova, Evgenia Kashina and Kirill Oseledets. Their friendship has supported me throughout these years.

I am grateful to my wife Marina for her love, encouragement and understanding. It is her support that gave me the strength to complete this work.

I am indebted to my parents Vladimir Vasilievitch Leontiev and Alla Davidovna Leontieva for their love and support during this work and throughout my entire life. This thesis is for them.

Contents

1 Introduction **1**
- 1.1 Overview . . . 1
- 1.2 Scope and contributions . . . 2
- 1.3 Organization . . . 3

2 Requirements and Type System Survey **4**
- 2.1 Terminology . . . 4
- 2.2 Essential features of a type system . . . 7
- 2.3 Object-oriented programming language requirements . . . 7
- 2.4 Database programming language requirements . . . 9
- 2.5 Object-oriented database system requirements . . . 10
- 2.6 Summary of requirements . . . 11
- 2.7 Related work . . . 19
  - 2.7.1 Type systems review . . . 21
- 2.8 Conclusions . . . 34

3 The Type System Design **36**
- 3.1 Types and behaviors . . . 37
  - 3.1.1 The notion of a type . . . 37
  - 3.1.2 Subtyping . . . 38
  - 3.1.3 Interface specification and product types . . . 41
  - 3.1.4 Behavior types . . . 46
  - 3.1.5 Mutable types . . . 50
  - 3.1.6 Parametric types and behaviors: specification . . . 53
  - 3.1.7 Parametricity and subtyping . . . 56
  - 3.1.8 Parametricity and interface specification . . . 64
  - 3.1.9 Union and intersection types . . . 65
  - 3.1.10 Constrained specifications . . . 66
  - 3.1.11 Subtyping and inheritance . . . 69
  - 3.1.12 Conclusions . . . 69
- 3.2 Classes and functions . . . 70
  - 3.2.1 Classes . . . 70
  - 3.2.2 Subclassing and class extension . . . 73
  - 3.2.3 Object creation and extent maintenance . . . 76
  - 3.2.4 Functions and associations . . . 78
  - 3.2.5 Dispatch . . . 80
  - 3.2.6 Closures and **typeif** . . . 83
- 3.3 Implementation types . . . 85
- 3.4 Three-layer design: Advantages and applications . . . 86
- 3.5 The basic type system . . . 87
  - 3.5.1 Object types . . . 87
  - 3.5.2 Atomic types . . . 89

3.5.3 Collection types 92
3.6 Conclusions 94

4 The Language and Typechecking **96**
4.1 Syntax and translation 96
4.1.1 The target language 96
4.1.2 The translation 100
4.2 Consistency 110
4.2.1 Notation 110
4.2.2 Local monotonicity 116
4.2.3 Simplicity and the user type graph 117
4.2.4 Type expansion 120
4.2.5 Constraint consistency 122
4.2.6 Global behavior consistency 124
4.2.7 Functional consistency 126
4.2.8 Dispatch consistency 131
4.3 Entailment 136
4.4 Conclusions 141

5 The Type System Theory **142**
5.1 Decidability 142
5.1.1 Finite evolution 143
5.1.2 Termination of expansion algorithm 146
5.1.3 Termination of flattening algorithm 147
5.1.4 Termination of entailment algorithm 149
5.2 Types 149
5.2.1 Regular trees 150
5.2.2 Subtyping 151
5.2.3 Entailment 153
5.2.4 Properties of entailment and flattening algorithms 156
5.2.5 Constrained types 160
5.3 Subject reduction 162
5.3.1 Static subsumption 162
5.3.2 Natural semantics 169
5.3.3 Execution state typing 170
5.3.4 Dispatch correctness 173
5.3.5 Subject reduction 178
5.4 Imperative types 195
5.5 Extensions 198
5.5.1 Errors, nulls, and exceptions 198
5.5.2 Local variables and dynamic scoping 199
5.5.3 Non-local returns 201
5.5.4 Handling object creation 201
5.6 Discussion 201
5.6.1 Extensibility 201
5.6.2 Type system evolution 202
5.6.3 Engineering trade-offs 204
5.6.4 Summary 205

6 Conclusions **206**
6.1 Summary and contributions 206
6.2 Future research 207

Bibliography **209**

A The Basic TIGUKAT Type System **218**
A.1 Common behaviors 218
A.2 Product types 218
A.3 Functional types 219
A.4 Object types 219
A.4.1 Special types 219
A.4.2 Atomic types 219
A.4.3 Collection types 222
A.4.4 Imperative types 224
A.4.5 Metatypes 225
A.4.6 Class types and the metaclasses 226

B Implementation Notes **227**
B.1 Object identifiers 227
B.2 Multiple dispatch 228
B.3 Implementation types and functions 228
B.4 Adding persistence 229

C Module System **230**
C.1 Information hiding and sharing 230
C.2 Export and import 231
C.3 Inner modules 232
C.4 Name search 233
C.5 Adding persistence 237

List of Tables

2.1 Type system index 20
2.2 Type system features 21
2.3 Type system expressibility 22
2.4 Type system tests 23

4.1 Variance combination 114

List of Figures

3.1 Type system layers 36
3.2 Person type hierarchy 41
3.3 Stream types. 62
3.4 Comparable types. 63
3.5 Class type hierarchy 78
3.6 Shape type hierarchy 82
3.7 TIGUKAT types. 87
3.8 TIGUKAT object types. 88
3.9 TIGUKAT atomic types. 89
3.10 TIGUKAT collection types. 93

4.1 Example user type graph $\mathcal{G}$ 120
4.2 Typing rules for the target language 128

5.1 Natural semantics of the target language 171
5.2 Typing rules for rt-objects 171

C.1 Module data structures 233
C.2 Name search 234
C.3 Name search in the given module 235
C.4 External name search in the given module 236

Chapter 1

Introduction

1.1 Overview

From its early days, object orientation (OO) was considered one of the most influential and useful programming paradigms. Its impact on research in virtually all areas of computing science can only be compared to that of relational algebra, or that of the functional and logic programming paradigms.

Much of the power of object orientation lies in the fact that it provides conceptual and modeling capabilities that allow it to express real-world entities with relative simplicity. Another source of the appeal of object orientation is its support for incremental software construction, provided in the form of code reuse.

All these advantages were first realized and exploited by the programming language (PL) research community. Starting with Simula-67 and Smalltalk, the development of object-oriented languages has become a landmark of programming language research in the last two decades.

Another major research area that experienced the impact of the object-oriented paradigm is the area of database management systems (DBMS). The high modeling power of object orientation along with its abstraction capabilities can make it one of the preferred paradigms in the development of DBMSs designed to deal with complex data-intensive applications, such as CAD/CAM, multimedia, and office information systems.

As both programming language and database system research areas experienced the impact of the object-oriented paradigm, so did their child — the area of database programming languages. This relatively young research area is still a source of prolific research and development activity, much of this activity being concentrated on object-oriented database programming languages or database programming languages with object-oriented features.

Object-oriented database programming languages (OODBPLs) have the potential to combine the modeling and software construction power of the object-oriented paradigm, extensive and efficient data storage and retrieval techniques of the modern database systems, and the efficiency and power of today's programming languages in a single uniform framework.

However, OODBPLs have yet to live up to their potential. Most of the modern OODBPLs are simply object-oriented programming languages with the concept of persistence added to them. They do not provide the full power of database systems either in their data access mechanisms or in their query capabilities. Therefore, the problems related to the design of an OODBPL that can combine the power of its three constituent parts (OO, DB, and PL) are nowadays the topics of extensive research activity.

Both the modeling and software construction powers of object-oriented languages are rooted in their type and inheritance systems. A properly designed, rich, and theoretically sound type system can greatly increase the power of a language, while a poor, inflexible type system can render almost all power inherent in the object-oriented paradigm useless.

While the type system of an object-oriented language greatly affects its characteristics, the type system of an OODBPL affects its characteristics even more. The reason for that is the presence of different and sometimes contradictory requirements that are imposed on the type system by an OODBPL's database and programming language components.

The presence of two sets of requirements makes the development of a type system for an OODBPL a challenging task. It, therefore, comes as no surprise that no existing type system satisfies the requirements that are imposed on OODBPLs.

While the task of developing such a type system is difficult, it is also quite rewarding. Apart from the proof of the validity of the very concept of OODBPL, the solution to this problem will provide the designers of OODBPLs with a consistent and uniform framework that would greatly facilitate the development of such languages in the future.

1.2 Scope and contributions

In this dissertation, a type system for an object-oriented programming language is developed. This type system satisfies a set of requirements that are placed on the type system by both database and object-oriented language components. These requirements are compiled from several sources and reviewed in terms of their relevance and necessity.

An extensive review of existing type systems is conducted in order to find out which of them satisfy the requirements. It is concluded that none of the reviewed type systems completely satisfies all the requirements and therefore the task of designing such a system is relevant and important.

The discussion of the type system presented in this dissertation is broken down into three components: the design principles, the typechecking and verification algorithms, and the theoretical framework that consists of the proofs of correctness of the presented techniques.

One of the design principles used in the construction of the presented type system is the novel principle of three-layered language design. The three layers are the interface layer, the implementation (code) layer, and the representation (data) layer. This design principle provides additional flexibility in specification and use of types, and facilitates better code reuse. A major motivation for the layered design is understandability of the type system by programmers. The system can be used at all the three layers; the deeper the layer, the more power it provides, and the greater complexity it involves. Other major design principles are substitutability, parametricity, verifiability, and soundness. The *source language* showing the use of this design and the type system is also presented. This language can serve as a core for the development of a full-fledged OODBPL.

Typechecking and verification algorithms are based on a notion of *type constraint entailment*. A program to be verified is transformed into a set of subtyping constraints, and the task of the type checker is to verify that these constraints admit a solution under certain assumptions. Another important aspect of program verification is the validity of the dispatch mechanism. Dispatch is the process of dynamically choosing the function to execute at a call-site depending on the types of actual arguments. The validity of dispatch is also tested by verifying entailment relationships between sets of subtyping constraints. Development of a correct and decidable typechecking algorithm for a type system as complex as the one described here is one of the major contributions of this work.

In order to formally prove the properties of the presented algorithms, a theory of subtyping and the natural semantics of a simplified version of the language (*the target language*) are developed. Equipped with this theory, proofs of decidability and correctness of the presented algorithms are given.

Thus, properties of the presented type system include verifiability, soundness, and expressibility. Verifiability is understood as a provable termination of all the algorithms involved in the program verification process. Soundness stands for the guarantee that a successfully typechecked program does not generate type errors at run-time. Expressibility is the ability of the type system to express various program and type structures. The result of this research is applicable to the development of persistent object-oriented type systems and languages and their theoretical underpinnings.

Other research areas relevant to the development of an OODBPL but not discussed in this dissertation include built-in object-oriented query language and its optimization, the persistence model, and the implementation of the language.

1.3 Organization

The remainder of this dissertation is organized into five chapters. Chapter 2 presents the set of requirements placed on the type system. The requirements are discussed and a test suite of programs designed to test the conformance of the type system to the expressibility and reflexivity requirements is developed. An extensive review of existing and proposed type systems is conducted. The chapter concludes with an evaluation of these type systems relative to the requirements. Chapter 3 is devoted to the description of the design principles and decisions made during development of the presented type system. It highlights various design choices and explains the ones that were made. The language and the typechecking and verification algorithms are presented in Chapter 4. Chapter 5 deals with the formal aspects of the presented type system. The proofs of the decidability and correctness results as well as the natural semantics of the simplified *target language* used for typechecking are presented in this chapter. Finally, Chapter 6 concludes with a summary of contributions and outlines directions for future research.

Appendix A presents an example of usage of the concepts developed in this dissertation for an OODBPL based on a uniform, behavioral object model [Pet94]. Appendix B provides a set of implementation notes and design decisions made during a preliminary implementation of a compiler and the run-time system for the TIGUKAT persistent programming language. Appendix C presents a module system design for the language. While not directly related to the type system itself, the module system provides support for additional information hiding and sharing.

Chapter 2

Requirements and Type System Survey

The purpose of this chapter is to answer two questions: "What are the requirements that a modern type system for an object-oriented database programming language should satisfy?" and "Are there any type systems developed to-date that satisfy these requirements?".

In order to answer the first question, the set of requirements put forth in the literature for type systems (Section 2.2), modern object-oriented programming languages (Section 2.3), database programming languages (Section 2.4), and object-oriented database systems (Section 2.5) by the leading researchers in these areas will be considered. Then, the set of required type system features will be extracted. The resulting combination of requirements is presented in Section 2.6. Also presented is the test suite that is used later to evaluate the existing type systems reviewed in this chapter.

Section 2.7 presents an extensive review of more than 30 languages and type systems. These type systems are evaluated with respect to the requirements and the test suite presented in Section 2.6. The result of this extensive analysis shows that while each of the requirements is satisfied by at least one type system, no type system satisfies all of them. It also enables the identification of the mechanisms that lie behind the strengths and weaknesses of the current type systems. The knowledge obtained this way will be used in subsequent sections to aid in the design and development of the type system that satisfies all of the above requirements, which is the primary goal of this work.

2.1 Terminology

Let us start by establishing some terminology to be used throughout the rest of this chapter. This is necessary as many of the terms used in the object-oriented language research area have no clear definition and are used differently by different authors. Most of the terminology below[1] comes from [Car89, BP94].

Object: A primitive term for a data item used to model a concept or a real-world entity.

Message: A part f of an invocation $x.f$; an identification for a related set of methods.

Function name: A part f of invocation $f(x)$; also can be termed as a message not associated with any type. Sometimes called a *free-floating function*.

Method: A procedure to be executed when an object is sent an appropriate message.

Function: A procedure to be executed when a function invocation is requested. Function relates to a function name the same way a method relates to a message.

[1] These definitions are not necessarily universally accepted, but they are used consistently throughout this dissertation.

Interface type: A description of messages applicable to an object.

Implementation (representation) type: A description of an object's structure.

Type: A shorthand for interface or implementation type or both, depending on the context.

Mutable object type: A type of an object that is capable of changing its state at run-time. For example, variables and arrays are mutable objects. These types are sometimes called *imperative types*.

Parametric (parameterized) type (message, function): A type (message, function) that describes a family of types (messages, functions) by using (an) explicit parameter(s). For example,

```
type T_List(X) {
      getAt(T_Integer): X;
};
```

Constrained type: A parametric type that places a constraint on its parameter(s). The mechanism that allows a programmer to specify such a type is called *bounded quantification*. For example

```
type T_PersonList(X) where X subtype of T_Person;
```

specifies a constrained type `T_PersonList`.

Intersection (greatest lower bound) types: An intersection of a set of types is a type that represents the greatest lower bound of the set in the type lattice. Intersection types are useful for typing of set-theoretic operations and queries.

Inheritance: A mechanism for making one (interface or implementation) type from another. A *single inheritance* requires that a type has at most one immediate ancestor (parent) in the inheritance chain; *multiple inheritance* lifts this restriction.

Interface subtyping: A partial order on interface types. An interface type A is a subtype of another interface type B when an object of the interface type B can be thought of as an object of the interface type A (e.g., a type `T_Student` is a subtype of `T_Person` since every student can be thought of as a person).

Implementation subtyping (code reuse): A partial order on implementation types. An implementation type A is a subtype of another implementation type B if it is possible to use code written for A to manipulate objects that have the implementation type B.

First-class object: An object that is capable of receiving messages.

Non-first-class object (value): An immutable object that lacks the ability to receive messages. Traditionally, it is assumed that values have no interface, just a set of operations defined on them. In this dissertation, such a set of operations will be considered an interface of a value.

Primitive (atomic) type: A primitive type is a type of basic (primitive) system-defined objects or values (such as integers, reals, characters, etc.). Primitive types usually have a special status in non-uniform systems and languages.

Soundness of a type system: Inability of a successfully typechecked program to produce run-time type errors. Sometimes divided into *static* and *dynamic* soundness. The latter makes sense for languages with dynamic type checking and assures that run-time type errors will be caught.

Verifiability of a type system (decidable typechecking): The ability to *verify* that a program does not contain type errors with respect to a particular type system; equivalently, the presence of a decidable typechecking algorithm. Note that verifiability does not imply soundness. For example, the language Eiffel [Mey88] has a decidable typechecking algorithm, but a successfully typechecked program can produce run-time type errors.

Substitutability: A property of a type system (language) that guarantees that an object of a subtype can be used everywhere the object of its supertype can (e.g. if a type **T_Person** is a supertype of **T_Student**, then a student can be legally used wherever a person can be).

Dispatch: A process of finding out at run-time which *method* of a particular *message* to execute. *Single dispatch* bases its decision on the type of the first argument (receiver) only, while *multiple dispatch* takes into account types of other arguments as well. The term *static dispatch* will be used to refer to a compile-time process of method determination.

Static/dynamic typing (typechecking): Static typechecking is done at compile-time, while dynamic typechecking is done at run-time.

Implicit/explicit typing: Languages with explicit typing require the programmer to insert type annotations in the program. Languages with implicit typing *infer* the types of expressions without any help from the programmer. Intermediate approaches are also possible.

Inclusion polymorphism: This term refers to a combination of subtyping and substitutability.

Parametric polymorphism: Parametric polymorphism is present when parametric specifications (e.g., parametric types) are supported.

Covariance (contravariance, novariance): Covariance means that changes in a particular type are parallel to the direction of the type hierarchy. In the following example the result type of the method **getAt** changes *co*variantly, as it is **T_Person** in definition 1 (which occurs in the type **T_PersonList**) and **T_Student** (a *subtype* of **T_Person**) in definition 2 (which occurs in the type **T_StudentList**, a *subtype* of **T_PersonList**).

```
type T_Student subtype of T_Person;

type T_PersonList {
      getAt(T_Integer): T_Person;  // 1
};

type T_StudentList subtype of T_PersonList {
      getAt(T_Integer): T_Student; // 2
};
```

The reverse direction is termed as *contravariant*. *Novariance* forbids any type changes along the type hierarchy. In this example, the argument type **int** changes novariantly, i.e. does not change at all.

The terms *class* and *subclassing* are deliberately avoided in this chapter as they are too overloaded. In object-oriented literature, the term *class* is used to denote a mechanism for object construction, an equivalent of an implementation type, an equivalent of an interface type, a set of objects satisfying a particular condition, or some combination of the above four notions. The definition of *subclassing* is equally overloaded.

Also, the term *type* is qualified as being either interface type or implementation type. This will prove useful when considering type systems where the two type notions are distinct.

2.2 Essential features of a type system

The major goals of a type system in today's programming languages and database systems include [Car89, Car97, BP94]:

1. Provide a programmer with an efficient way of catching programming errors *before* a program (or a part of it) is actually executed. This is often considered to be the major objective of a type system in the programming language community.
2. Serve as a data structuring tool for design and modeling purposes. Many design technologies that have emerged through the past decade rely partially or fully on type systems to provide a convenient design and documentation framework for a system development process. This is especially true of object-oriented design technologies. This is often considered to be the major goal of a type system in the database community.
3. Provide a convenient framework for program maintenance. This includes documentation at the production stage of program evolution as well as the ability of a programmer to understand the functionality and interfaces of a completed product.
4. Provide sufficient information for optimization purposes. The information provided by the type system can be used by an optimizing compiler, interpreter, or a query optimizer (in persistent systems) to improve the efficiency of a program.

In order for a compiler (or interpreter) to be able to typecheck a program (or a part of it), there must exist a *typechecking algorithm*[2]. Existence of such an algorithm for a given type system is termed as *verifiability* of the type system. Thus, a type system should be *verifiable*. It is preferred that a type system be *decidably verifiable*; however, one may have to put up with an undecidable type system just as one puts up with undecidable programs if enough expressive power is desired [BP94]. The verifiability property also implies that a type system should be *provably sound*, i.e. there should exist a formal proof that a successfully type-checked program does not generate any type errors at run-time.

If a compiler finds a type error and reports it to a programmer, the latter should have sufficient information to be able to understand the reason for the type error in order to correct it. Thus, the type system should be *transparent*.

A type system should also be *enforceable* in order to prevent an execution of type-incorrect programs. This implies that programs have to be written with as much type information as possible to prevent "false alarms".

Finally, a type system should be *extensible*. This requirement stems from the fact that none of the existing type systems were found to be satisfactory for all possible applications. Therefore the chance that any new type system will satisfy *all* application domains is remote. If a type system can not be extended, it will sooner or later be abandoned for a type system that can adapt to new application requirements. Switching from one type system to another is extremely costly both in terms of people resources (that have to be reeducated) and in terms of data conversion costs.

2.3 Object-oriented programming language requirements

Pure object-oriented programming languages pose some specific requirements on their type systems. These requirements will be constructed by considering features essential for pure object-oriented languages and reformulating them in terms of type system features.

To be called *a pure object-oriented language*, a language should possess at least the following properties[3] (most of the following is borrowed from [TNG92]):

[2]Much of the discussion below is borrowed from [Car97].

[3]Some of the features listed here are advocated in [TNG92] as absolutely necessary for future object-oriented languages.

1. Encapsulation. This property is usually considered as one of the characteristic features of object-oriented languages and greatly facilitates code reuse. It refers to the ability of a language to shield internals of an object implementation from outside access.

2. Inheritance. This is a characteristic property of object-oriented languages as well. The inheritance mechanism promotes and facilitates well-structured software design and reusability of the code. Multiple inheritance is highly desirable, as its absence leads to clumsy or limited type specification in some important cases.

3. Uniformity. Primitive values (integers etc.), types, and messages (methods) should be first-class objects. If this requirement is not satisfied, the language will have to handle the non-object entities in some non-object-oriented way, and will therefore not be *purely* object-oriented. Note that the existence of methods that are able to operate on types and other methods is a consequence of this property. This is also advocated in [Hau93].

4. Object access and manipulation uniformity. An object can only be manipulated by methods defined for it. Together with uniformity, this property provides for purely object-oriented programming.

5. Method uniformity. This refers to the absence of distinction between *stored* and *computed* methods or, equivalently, the absence of public instance variables. This requirement is important as its violation breaks encapsulation and may effectively hinder the usefulness of the code reusability provided by inheritance.

6. Separation between interface and implementation inheritance (sometimes termed as separation between type and class hierarchies). This is actually a consequence of encapsulation, inheritance, and object manipulation uniformity. Additional arguments in favor of such separation can be found in [LP91, Cas96, Tai96, BLR96, LÖS98].

7. Multi-methods (multiple dispatch). This refers to the ability of a language to use types of all arguments during dispatch. Traditionally, only the type of the first argument (the receiver) is used. This property of the language is essential to adequately model binary methods [BCC+96, Cas96, LM98, FM96] and certain object-oriented design patterns [BLR96].

Using these requirements for pure object-oriented languages, it is now possible to formulate desirable features of type systems for such languages. These requirements are:

1. Inheritance mechanisms for both interface and implementation inheritance. This requirement is a direct consequence of the language requirements 2 and 6 above.

2. Type system reflexivity. This is necessary to ensure uniformity of the language (requirement 3), since types (classes) have to be objects. Since every object has a type, types and classes need to have types as well. Thus, the type system needs to be reflexive.

3. Method types. This is also a consequence of uniformity. Indeed, since methods have to be objects to ensure uniformity, they will have types. Moreover, when methods are manipulated as objects (e.g., passed as arguments to other methods), their types should be descriptive enough to ensure the validity of type-checking.

4. Method uniformity at the type level (no distinction between types of *stored* and *computed* methods). This is a direct consequence of the language requirement 5.

5. Support for multi-methods (multiple dispatch). This is a consequence of the language requirement 7.

Another desirable feature of an object-oriented type system is substitutability (at least for interface inheritance). This property is essential to achieve one of the primary goals of the object-oriented paradigm: code reuse.

2.4 Database programming language requirements

Database programming languages (DBPLs) possess their own set of distinguishing features that poses additional requirements on type systems. The approach of the previous subsection will be used to derive the type system features from the following list of necessary features of a persistent language that is taken[4] from [AB87, AM95].

1. Persistence independence (the form of the program is independent of the longevity of data the program operates upon). This is necessary to provide seamless integration between the database and the language and to significantly reduce the amount of code necessary to deal with persistent data.

2. Orthogonality of type and persistence (data of all types can be persistent as well as transient). This is an aid to data modeling as it ensures that the model can be independent of the longevity of data. It also eliminates the need of explicit persistent-to-transient data conversions.

3. User-defined type constructors. This requirement is due to the necessity of modeling new, potentially complex data structures in a uniform and consistent manner.

4. Information hiding (also known as encapsulation). Encapsulation allows for data modeling at a higher (more abstract) level as it hides the implementation details and gives the programmer an ability to deal with abstract interfaces rather than concrete data structures. It greatly facilitates modeling, code reuse, and component integration.

5. Polymorphism (parametric, inclusion, or both). Parametric and inclusion polymorphisms make the specifications more succinct and precise. They also allow for a significant reduction in the amount of code that needs to be written to specify and implement a particular data model, as a significant portion of the specifications are reused via genericity achieved by the use of polymorphic constructs.

6. Static and strong typing with provisions for partial type specification (which necessitates the presence of a type mechanism similar to one formed by type constructor **dynamic** and operator **typecase**). This is necessary in order to deal with data that come from a persistent store whose structure is only partially known at the time the program is written. An example of a program that requires such capabilities is a generic database browser that is supposed to work on any database independently of its structure.

7. Incremental program construction and modularity. This principle ensures that most of the program modifications can be done locally, without affecting the rest of the code. While this property is very important for programming languages in general, it is even more important for database programming, since databases tend to exist and evolve for extensive periods of time. As a database evolves, the programs designed to operate on it have to evolve as well. Modularity is one of the major features that significantly reduce the overhead of such an evolution.

8. Query facilities. One of the main reasons behind the success of the relational data model was its ability to support declarative, simple, yet powerful data access/query languages. In order for object systems to be successful, they must provide querying capabilities equal or exceeding those of the relational systems.

9. Ability of a program to deal with state change. This requirement is necessary as persistent data outlive the program and if a program is not able to change data, the state of persistent data will never change.

From this list of requirements, the following properties of the type system can be derived:

[4]Features not related to type system are dropped from the list.

1. Types (classes) in the type system should not be specified as either persistent or transient. This is the DBPL requirement 2 reformulated in terms of type system terminology.

2. User-defined type constructors. This is the same as the DBPL requirement 3.

3. Encapsulation. This follows from the DBPL requirement 4.

4. The presence of parametric types. This follows from the DBPL requirements 5 (parametric polymorphism) and 3. It is also desirable to handle bounded (constrained) parametric types as it increases the modeling power of the type system.

5. Possibility of partial type specification and dynamic type checking. This follows from the DBPL requirement 6.

6. Verifiable and sound type system (as a consequence of the DBPL requirement 6).

7. Incremental type checking (as a consequence of the DBPL modularity requirement).

8. The ability of the type system to correctly type declarative queries. This stems from the DBPL requirement 8. According to [BP99], this requires that the type system can support union (least upper bound) types.

9. The ability of the type system to deal with types of mutable objects (later referred to as mutable object types) and assignment. This is a direct consequence of the DBPL requirement 9.

In addition to the above, [KCMS96] advocates the use of reflection in persistent object systems.

The combination of object-orientedness and persistence poses some additional requirements described in the next subsection.

2.5 Object-oriented database system requirements

A list of features needed or desirable in object-oriented database management systems (OODBMS) and the rationale behind it are given in [ABD+92]. This is the most comprehensive of such lists published so far.

The following list[5] is the part of it that is related to type system issues. It contains the additional requirements to those already listed in the previous subsections.

1. Complex objects (orthogonal type constructors should include at least sets, tuples, and lists). This is necessary to ensure that the modeling power of the system is sufficient to deal with modern applications, such as CAD/CAM, medical and geographical information systems.

2. Extensibility: user-defined and system types should have the same status and the user should be able to add new "primitive" types to the system. This is also due to the necessity of dealing with new demanding application areas. It is impossible to anticipate all the data types that will be required to model the data structures in those areas since some of them are yet to emerge. By providing the same status to system and user-defined types, the system guarantees that its capabilities are not decreased when it is applied to a new application domain. Also advocated in [MS91].

3. Views. A view is, in a sense, an ability to transparently change the appearance of data for different users (clients). The importance of this concept as well as its usefulness and power have been convincingly demonstrated by years of experience obtained by the research and industrial communities in dealings with relational databases.

[5]Some issues often considered as deficiencies of object-oriented systems (for example, in [Kim93]) but deemed optional in [ABD+92] are listed here as mandatory. The reason for that is the understanding that if object-oriented databases are to be the next step in the database development, they should utilize the advances already made in relational databases.

4. Dynamic schema evolution. This requirement is based on the necessity to maintain (and change) the database structure over extensive periods of time. While it is sometimes possible to create a completely new database with a new schema and migrate data to it, this approach is usually quite expensive and results in a substantial down time. This is often not acceptable in large distributed applications such as air traffic control systems. Dynamic schema evolution makes it possible to change a database structure transparently to its users.

These additional requirements have to be taken into account when designing a type system for a pure object-oriented database programming language. Next, these requirements will be summarized and a short overview will be given.

2.6 Summary of requirements

The following is the compilation of all type system requirements presented so far. The categorization of the requirements presented here is subjective, but it does provide a useful structure to the extensive set of requirements compiled so far. Each requirement listed below contains a reference to the section where it has been introduced and explained.

1. Theoretical requirements
   (a) Verifiability (preferably decidable) and provable soundness of the type system. These features are necessary for the type system to be useful for program verification (see Section 2.2 and Section 2.4).
2. Inheritance requirements
   (a) Inheritance mechanisms for both interface and implementation inheritance (Section 2.3).
   (b) Substitutability property (at least for interface inheritance) (Section 2.3).
3. Expressibility requirements
   (a) Method types (Section 2.3).
   (b) Parametric types (Section 2.4).
   (c) Orthogonal type constructors (at least sets, tuples, and lists) (Section 2.4).
   (d) Encapsulation (Section 2.3 and Section 2.4).
   (e) The ability of the type system to deal with mutable object types and assignment (Section 2.4).
   (f) The ability of the type system to correctly type multi-methods (Section 2.3).
   (g) The ability of the type system to correctly type SQL-like queries (Section 2.4).
4. Uniformity requirements
   (a) Extensibility (user-defined and system types should have the same status) (Section 2.5).
   (b) Types (classes) in the type system should not be specified as either persistent or transient (Section 2.4).
   (c) Method uniformity at the type level (no distinction between types of *stored* and *computed* methods) (Section 2.3).
5. Reflexivity requirements
   (a) Type system reflexivity (Section 2.3).
6. Dynamic requirements

   (a) Possibility of partial type specification and dynamic type checking (Section 2.4).
   (b) Provisions for schema evolution (Section 2.5).

7. Other requirements

   (a) Transparency of the type system for a programmer (Section 2.2).
   (b) Incremental type checking (Section 2.4).
   (c) The ability to define views in a type-safe fashion (Section 2.5).

Some of the above requirements are complimentary, while others are contradictory. Most notably, decidable verification conflicts with reflection (as shown, for example, in [Car86b]). Also, enforceability conflicts (to a degree) with partial type specification. Another conflict can be seen in that the complexity of the type system that satisfies the expressibility requirements above will most probably make the resulting type system much less transparent for a programmer than one would like it to be. [CMM91] also identified a conflict between substitutability, mutable types, and static type safety. The presence of such contradictory requirements makes the task of designing a type system that satisfies them particularly difficult.

The following is a set of test programs that a type system should be able to type correctly[6]. These tests are primarily designed to test type systems for their expressibility as this is the most difficult set of requirements to check; however, the last test is the test for reflexivity and uniformity. The programs are written in an object-oriented pseudo-language[7].

These programs are designed to test known problem areas of object-oriented type systems. They are also used to verify the ability of the type system to consistently and orthogonally combine parametric and inclusion polymorphism with mutable types and assignment. This has to be done because soundness, verifiability, parametricity, substitutability, and mutable types are all among the requirements for an OODBPL type system.

Many of the expressibility tests are adapted from [BP94] which presented a benchmark for testing type system expressibility. However, their benchmark was designed to measure the expressibility of a type system for an object-oriented programming language and not for an object-oriented *database* programming language. Thus, some tests related to the additional expressibility requirements presented above were added.

The requirement related to mutable object types and assignment is tested by each of the tests below. This is done in order to verify that a type system can deal with mutable types in combination with parametric and inclusion polymorphisms. This is a well-known problem area in object-oriented type systems [CMM91].

1. Types **T_Person** and **T_Child** with method **getAge** that returns **T_Integer** when applied to a person and **T_SmallInteger** when applied to a child. *(PERSON)*

```
type T_Integer;
type SmallInteger subtype of Integer;

type T_Person {
      getAge(): T_Integer;
};
type T_Child subtype of T_Person {
      getAge(): T_SmallInteger;
};

T_Person p := new T_Person (...);
T_Child  c := new T_Child (...);
```

[6]Note that the terms "test" and "test program" are not used here in the traditional software engineering sense.
[7]The programming language examples are used to illustrate the tests and not to suggest the necessary language constructs.

```
T_Integer i;
T_SmallInteger si;

i := p.getAge();            // should be Ok
i := c.getAge();            // should be Ok

si := p.getAge();           // should cause compile-time error
si := c.getAge();           // should be Ok
```

This test is designed to verify that subtyping does not necessitate the absence of changes. Surprisingly, there is a considerable number of languages that do not allow *any* changes while inheriting, only additions. This significantly limits the power of the type system and forces the designer to use less specific type information.

2. Types `T_Point` and `T_ColorPoint`, with equality on both. The equality between color points should take color into account, while the equality between two points or between a point and a color point should ignore it. *(POINT)*

```
type T_Point {
    equal(T_Point p):T_Bool implementation  ... ;      // equal1
};
type T_ColorPoint subtype of T_Point {
    equal(T_ColorPoint p):T_Bool implementation  ... ; // equal2
};

T_Point p1 := new T_Point (...);
T_ColorPoint p2 := new T_ColorPoint (...);

p1.equal(p1);                 // should call equal1
p1.equal(p2);                 // should call equal1
p2.equal(p2);                 // should call equal2
p2.equal(p1);                 // should call equal1

p1 := new T_ColorPoint (...);

p1.equal(p1);                 // should call equal2
p1.equal(p2);                 // should call equal2
p2.equal(p2);                 // should call equal2
p2.equal(p1);                 // should call equal2
```

This test is adapted from [BP94]. It tests the type system's ability to deal with *binary methods*, a well-known problem area in object-oriented type systems [BCC+96, FM96].

3. Types `T_Number` (with unrelated subtypes `T_Real` and `T_Radix`) and `T_Date` with comparison methods such that comparing two numbers or two dates is legal, while their cross-comparison is not. All method code below, except for the code for the method `less`, should be reused. *(COMPARABLE)*

```
interface I_Comparable {
    less(selftype c):T_Bool;
    greater(selftype c):T_Bool  implementation { return c.less(this); };
};
type T_Number implements I_Comparable {
    less(T_Number n):T_Bool   implementation  ... ;   // less1
};
type T_Real subtype of T_Number { ... };
```

```
    type T_Radix subtype of T_Number { ... };
    type T_Date implements I_Comparable {
        less(T_Date d):T_Bool     implementation  ... ;   // less2
    };

    T_Date d1, d2;
    T_Number n1, n2;
    T_Radix radixVar := 0xF;
    T_Real realVar := 1.0;

    d1 := '2/3/99'; d2 := '3/4/78';

    n1 := realVar;
    n2 := radixVar;

    n1.less(n2);                   // should call less1
    n2.greater(n1);                // should eventually call less1

    n1.less(realVar);              // should call less1
    n2.greater(radixVar);          // should eventually call less1

    d1.less(d2);                   // should call less2
    d1.greater(d2);                // should eventually call less2

    d1.less(n1);                   // should cause compile-time error
    n1.less(d1);                   // should cause compile-time error
```

This is an additional test for binary method handling. It tests whether the subsumption property can be maintained in the presence of binary methods. This test is necessary due to the fact that some approaches to the binary method problem (most notably, matching [BSG95, BFP96]) abandon substitutability, and substitutability is one of the requirements for an OODBPL type system.

4. Parametric input/output/IOstream type hierarchy such that the three types are parameterized by the type of objects readable/writable to/from a particular stream, with **T_IOstream** being a subtype of both input and output stream types. *(STREAMS)*

```
    type T_InputStream(covar X) {
        get():X;
        isEmpty():T_Bool;
    };
    type T_OutputStream(contravar X) {
        put(X arg);
    };
    type T_IOStream(novar X) subtype of T_InputStream(X), T_OutputStream(X);

    type T_Point { ... };
    type T_ColorPoint subtype of T_Point { ... };

    T_Point p := new T_Point (...);
    T_ColorPoint cp := new T_ColorPoint (...);

    T_InputStream(T_Point) isp;
    T_OutputStream(T_Point) osp;
    T_IOStream(T_Point) iosp;
```

```
T_InputStream(T_ColorPoint) iscp;
T_OutputStream(T_ColorPoint) oscp;
T_IOStream(T_ColorPoint) ioscp;

... // Initialization of the above streams

p := isp.get();                // should be Ok
p := iscp.get();               // should be Ok
cp := isp.get();               // should cause compile-time error
cp := iscp.get();              // should be Ok

osp.put(p);                    // should be Ok
oscp.put(p);                   // should cause compile-time error
osp.put(cp);                   // should be Ok
oscp.put(cp);                  // should be Ok

isp := iosp;                   // should be Ok
isp := ioscp;                  // should be Ok
iscp := iosp;                  // should cause compile-time error
iscp := ioscp;                 // should be Ok

osp := iosp;                   // should be Ok
osp := ioscp;                  // should cause compile-time error
oscp := iosp;                  // should be Ok
oscp := ioscp;                 // should be Ok
```

This test is designed to verify that a combination of parametric and inclusion polymorphism in the type system does not adversely affect either of them. In other words, it tests the orthogonality of the two polymorphisms. The presence of both parametricity and inclusion polymorphism (subtyping + substitutability) in the type system is among the requirements compiled earlier. The unrestricted combination of the two polymorphisms is known to be difficult [DGLM95, BOSW98c].

5. Sorting of arbitrary objects under the constraint that all the objects have a comparison method. *(SORT)*

```
interface I_Comparable {
    less(selftype c):T_Bool;
};
type T_Number implements I_Comparable { ... };
type T_Person { ... };
type T_List(novar X) { ... };

sort(T_List(X) list): T_List(X) where (X implements I_Comparable)
                implementation ... ;

T_List(T_Number) ln;
T_List(T_Person) lp;

... // Initialization of list variables

ln := sort(ln);                // should be Ok
lp := sort(lp);                // should cause compile-time error
lp := sort(ln);                // should cause compile-time error
```

```
    ln := sort(lp);                       // should cause compile-time error
```

This test is due to [BP94]. It is designed to verify the ability of the type system to deal with bounded quantification of the form "for all types satisfying a given condition". This is yet another aspect of the binary method problem. It also tests the ability of the type system to provide a link between parametricity and subtyping.

6. Generic sort with a comparison method (<) as a parameter. The generic sort can be applied to a set of any objects provided that an appropriate comparison method is supplied. (*GENSORT*)

```
type T_List(X) { ... };
type T_Number {
    less(T_Number arg):T_Bool;
};
type T_Date {
    compare(T_Date arg):T_Bool;
};

sort(T_List(X) list, (X,X):T_Bool comparison): T_List(X)
                implementation ... ;

T_List(Number) ln;
T_List(Date) ld;

... // Initialization of list variables

ln := sort(ln,less);            // should be Ok
ld := sort(ld,compare);         // should be Ok
ln := sort(ld,compare);         // should cause compile-time error
ln := sort(ln,compare);         // should cause compile-time error
```

This test is also from [BP94]. It is designed to verify that the type system is capable of combining parametricity, method typing, and substitutability.

7. A single-linked list node type and a double-linked list node type, where the second type inherits from the first one. A single-linked list node type is a recursively defined type with a mutable attribute that represents the list node linked to the given one. A double-linked list node type has an additional mutable attribute to represent the second link. The type system should not allow links between different node types.

Note that in this example the double-linked node type is *not* a subtype of a single-linked node type; however, the type system should be flexible enough to allow code reuse between them. The code sample uses the keyword **extends** in order to describe this relationship between types. (*LIST*)

```
type T_LinkedListNode {
    selftype next;
    getNext():selftype implementation { return next; };
    attach(T_LinkedListNode node)
             implementation  ... ;        // attach1
};
type T_DoubleLinkedListNode extends T_LinkedListNode {
    selftype prev;
    getPrev():selftype implementation { return prev; };
    attach(T_DoubleLinkedListNode node)
             implementation  ... ;  // attach2
};
```

```
T_LinkedListNode lln1, lln2;
T_DoubleLinkedListNode dlln1, dlln2;

... // Initialization of list node variables

lln1 := dlln1;                // should cause compile-time error
lln1 := lln2;                 // should be Ok
lln1.attach(lln2);            // should call attach1
dlln1.attach(dlln2);          // should call attach2
lln1.attach(dlln1);           // should cause compile-time error
dlln1.attach(lln1);           // should cause compile-time error
```

This test is also adopted from [BP94]. It checks whether the type system supports code reuse beyond that provided by subtyping. In other words, it checks if code reuse is possible between types that are not in a subtyping relationship with each other. Situations analogous to the one described in this test occur frequently when dealing with mutable types.

8. Set union and intersection for immutable sets. *(SET)*

```
type T_Set(X) {
    union(T_Set(Y) summand): T_Set(lub(X,Y));
    intersection(T_Set(Y) summand): T_Set(glb(X,Y));
};

type T_Person { ... };
type T_Student subtype of T_Person { ... };

T_Set(T_Person) sp1, sp2;
T_Set(T_Student) ss1, ss2;

... // Initialization of set variables

sp1 := sp1.union(sp2);        // should be Ok
sp1 := sp1.union(ss1);        // should be Ok
sp1 := ss1.union(sp1);        // should be Ok
sp1 := ss1.union(ss2);        // should be Ok

ss1 := sp1.union(sp2);        // should cause compile-time error
ss1 := sp1.union(ss1);        // should cause compile-time error
ss1 := ss1.union(sp1);        // should cause compile-time error
ss1 := ss1.union(ss2);        // should be Ok

sp1 := sp1.intersection(sp2);        // should be Ok
sp1 := sp1.intersection(ss1);        // should be Ok
sp1 := ss1.intersection(sp1);        // should be Ok
sp1 := ss1.intersection(ss2);        // should be Ok

ss1 := sp1.intersection(sp2);        // should cause compile-time error
ss1 := sp1.intersection(ss1);        // should be Ok
ss1 := ss1.intersection(sp1);        // should be Ok
ss1 := ss1.intersection(ss2);        // should be Ok
```

This test is designed to verify that set operations used in SQL-like declarative queries can be successfully and precisely typed. This is necessary to ensure seamless integration of such queries into the language.

9. Function *apply*. *(APPLY)*

```
apply((X):Y msg, X obj):Y implementation ... ;

type T_Integer { ... };
type T_SmallInteger subtype of T_Integer { ... };

type T_Person {
     getAge():T_Integer;
};
type T_Child subtype of T_Person {
     getAge():T_SmallInteger;
};

T_Person p := new T_Person (...);
T_Child  c := new T_Child (...);
T_Integer i := 1000;
T_SmallInteger si := 5;

i := apply(getAge,p);          // should be Ok
i := apply(getAge,c);          // should be Ok

si := apply(getAge,p);         // should cause compile-time error
si := apply(getAge,c);         // should be Ok
```

This test is analogous to the "λ-calculus" test presented in [BP94] (it differs in that this test also includes assignment). It is designed to verify the ability of the type system to deal with method types and uniformly treat methods as objects in the system.

10. General database browser. The browser should be able to deal with databases that have an unknown schema[8]. *(BROWSER)*

```
printNumber(T_Number num)  implementation ... ;
type T_Person {
     getAge():T_Integer;
};

T_Object root;
T_Database db;

db.open();

root := db.getRoot();
typecase root.typeOf() {
    subtype of T_Number: {
              printNumber(root); ...          }; // should be Ok
    subtype of T_Person: {
              printNumber(root.getAge()); ... }; // should be Ok
    otherwise: {
              print("Something else"); };
};

root.getAge();                        // should cause compile-time error
printNumber(root.getAge());           // should cause compile-time error
```

[8]The given code only tests the ability of the type system to deal with dynamic type information in a type-safe manner. A complete browser would also require the ability to examine type structure of previously unknown types.

This test checks the ability of a type system to handle partial type specifications and dynamic type checking. Both are on the list of requirements for an OODBPL type system.

This set of requirements and tests will be used in the next section to evaluate existing languages and type systems.

2.7 Related work

In this section, type systems of many current languages[9] as well as theoretical developments in the area will be reviewed. These type systems and languages are listed in Table 2.1. The table gives references and pages in this dissertation where a given system is reviewed.

The comparison between the type systems is presented in Table 2.2, Table 2.3, and Table 2.4. Not all of the languages and systems listed in Table 2.1 are present in these tables. Type systems that are superseded by other type systems reviewed in this section, languages that have no static type systems, and incomplete type systems are excluded from the review tables.

Table 2.2 lists features of the reviewed type systems that correspond to the requirements listed in Section 2.6. Verifiability is understood as the presence of a decidable type checking algorithm. Static soundness means that a successfully typechecked program does not produce errors at run-time, while dynamic soundness means that the program reports *all* possible type errors at run-time in a well-defined and predictable manner. Static soundness is strictly stronger then dynamic soundness. The column *uniformity/atomics* means that objects of primitive (atomic) types have the same rights as objects of user-defined types. The column *uniformity/methods* refers to the ability of a language to treat methods (or messages, or both) as objects. *Reflection/typecase* indicates the ability of a language to deal with dynamic type checking in a type-safe manner. Finally, *reflection/evolution* shows whether a given language supports incremental type system evolution.

Table 2.3 compares various aspects of type system expressibility. The first two columns indicate whether a given type system has a notion of subtyping, shows what kind of subtyping (structural (implicit) or user-defined (explicit)) it supports, and what kind of inheritance (single or multiple) the type system has. The third column shows whether the type system supports method (function) types. The fourth column addresses the issue of dispatch (single or multiple) in the given type system. Columns 5 through 8 deal with parametricity and its relationship with subtyping. Column 5 indicates whether a given type system supports parametric types. Column 6 shows if a type system can deal with constrained parametric types. Positive indication in column 7 means that a type system makes it possible for different parametric types formed using the same type constructor to have a subtyping relationship with each other (for example, `T_Set(T_Person)` is a subtype of `T_Set(T_Student)`). Column 8 indicates whether the type system is capable of specifying subtyping relationships between parametric types with *different* type constructors (e.g. `T_List(T_Person)` is a subtype of `T_Set(T_Person)`). Column 9 is an indication of the ability of a type system to deal with mutable types. This indication is negative for type systems of purely functional languages. Column 10 shows whether a type system supports intersection (greatest lower bound) types.

Finally, Table 2.4 demonstrates the performance of the reviewed type systems on the test suite.

From the analysis of the results presented in Table 2.4 it can be concluded that languages Mini-Cecil and Transframe rate the best on the test suite. However, none of them has provably sound typechecking; in fact, typechecking Mini-Cecil is likely to be undecidable [Lit98]. Of the systems with sound and verifiable type checking, the most impressive is Sather; however, its type system lacks multiple dispatch and union types. Soundness of Sather type system has not been formally proven. Almost all "theoretical" type systems show similar performance on the test suite.

Sound type checking, substitutability, parametricity, method types, and multi-methods appear together in only one type system: that of $\mathrm{ML}_{\leq}$. However, $\mathrm{ML}_{\leq}$ (as well as most ML clones) severely restricts usage of mutable types and does not deal well with certain aspects of binary methods and parametricity (failed tests *COMPARABLE* and *STREAMS*).

[9] Due to the enormous number of languages constantly being developed by the scientific community, this review has to be incomplete. However, every effort has been made to include known languages with interesting type system features (or their analogs). Thus I hope that none of the essential type systems are left out.

Table 2.1: Type system index

	References	Reviewed on page
C++	[Str91]	21
E	[RCS93]	23
O++	[AG89]	23
O_2	[LRV92]	24
Java 1.1	[AG96, DE97, Sar97]	24
Generic Java (GJ)	[BOSW98c, BOSW98a, BOSW98b]	24
Java parametric extension (JPE)	[MBL97]	24
Pizza	[OW97, AFM97]	24
Virtual types in Java (JVT)	[Tho97, BOW98]	25
ODMG/OQL 2.0	[CBB+97]	25
Modula-2	[Wir83]	25
DBPL	[SM94]	25
Modula-90	[LML+94]	25
Modula-3	[Har92]	26
Oberon-2	[MW93, RS97]	26
Lagoona	[Fra97]	26
Theta	[DGLM95]	26
Ada 95	[Ada95]	27
Eiffel	[Mey88]	27
Sather	[SO96]	27
Emerald	[RTL+91]	27
BETA	[MMPN93]	27
VML	[KT92]	28
Napier88	[MBC+96]	28
Smalltalk	[GR89]	29
Strongtalk	[BG93]	29
Cecil	[Cha93, Cha92b]	29
BeCecil	[CL96, CL97]	29
Mini-Cecil	[Lit98]	29
Transframe	[Sha97]	30
CLOS	[BDG+88]	30
Dylan	[App94]	30
TM	[BBdB+93]	30
ML	[MTH90, Wri93]	31
Machiavelli	[OBBT89, BO96]	31
Fibonacci	[AGO95]	32
$ML_{\leq}$	[BM96b, BM96a]	32
Constrained types in ML	[AWL94, AW93, Pot98, Reh98]	33
Constrained types in Erlang	[MW97]	33
$\lambda\&$-calculus	[CGL95]	33
PolyTOIL	[BSG95, BSG94]	33
Loop	[ESTZ95, EST95b, EST95a]	34
TL	[MS92, MMS94]	34
TooL	[GM96]	34
TOFL	[QKB96]	34

Table 2.2: Type system features

Type system	Verifiability	Soundness		Separation between interface and implementation	Substitutability	Uniformity		Reflection	
		Static	Dynamic			Atomics	Methods	Typecase	Evolution
C++	+	-	-	-	+[3]	-	+	-	-
E	+	-	-	-	+[3]	-	+	-	-
C++	+	-	-	-	+[3]	-	+	-/+[5]	-
O_2	+	Unknown	Unknown	-	-	-	-	-	+/-[7]
Java 1.1	+	-	Core only	+	+/-	-	-	+	-
GJ	+	-	Unknown	+	-	-	-	+	-
JPE	+	-	Unknown	+	-	-	-	+	-
Pizza	+	+[1]	+	+	-	+	+	+	-
JVT	+	-	Unknown	+	-	-	-	+	-
ODMG 2.0	+	Unknown	Unknown	+	-	-	-	+	-
DBPL	+	Unknown	Unknown	+[2]	n/a	n/a	+	-	-
Modula-90	+	Unknown	Unknown	+[2]	+[4]	-	+	+	-
Modula-3	+	Unknown	Unknown	+[2]	+	-	+	+	-
Oberon-2	+	Unknown	Unknown	+[2]	+	-	+	+	-
Theta	+	Unknown	Unknown	+	Unknown	-	+	+	-
Ada 95	+	Unknown	Unknown	+[2]	-	-	+	+	-
Sather	+	+	+	+	+	-	+	+	-
Emerald	+	Unknown	Unknown	+	+	+	-	+	+/-[7]
BETA	+	-	Unknown	+[2]	+	+	+	+	+/-[7]
Napier88	+	+	+	+/-	n/a	n/a	+	+	+
Strongtalk	+	Unknown	Unknown	+	-	+	+	+	+/-[7]
Cecil	+	-	Unknown	+	-	+	+	+/-[6]	+
BeCecil	+	Unknown	Unknown	+	Unknown	+	+	-	+
Mini-Cecil	Unknown	Unknown	Unknown	+	Unknown	+	+	-	-
Transframe	+	-	Unknown	+	Unknown	+	+	+	+
TM	+	Unknown	n/a	-	-	+	-	-	-
ML	+	+	n/a	+[2]	n/a	+	+	-	-
Machiavelli	+	+	n/a	-	+	+	+	-	-
Fibonacci	+	+	n/a	+/-	+	+	+	-	-
$ML_{<}$	+	+	n/a	-	+	+	+	-	-
PolyTOIL	+	+	n/a	-	+	+	+	-	-
Loop	+	+	n/a	+	+	+	+	-	-
TL	+	Unknown	Unknown	-	Unknown	+	+	+	-
TooL	Unknown	Unknown	n/a	+	Unknown	+	+	-	-
TOPL	+	+	n/a	-	+	+	+	-	-

[1] Except for covariant arrays which have been kept in Pizza only for backward compatibility with Java
[2] These features are based on *modules(packages, fragments)* rather than on *types*
[3] Substitutability works for pointers and references only
[4] For object types only
[5] Even though a type can be examined dynamically in O++, typechecking does not take this into account
[6] Classes (implementation types) only
[7] Evolution is type-unsafe

The test *STREAMS* proved to be the most difficult one. This is surprising as the test outlines the situation that occurs in almost every language dealing with I/O operations on a relatively high level. Only the type systems of Emerald and Mini-Cecil where able to pass this test; however, none of these type systems has a proof of soundness.

Overall, type systems with nice theoretical properties show only moderate performance on the test suite, while type systems that perform well on tests lack a theoretical basis.

The rest of this section presents a detailed review of various type systems. The index of these type systems is found in Table 2.1.

2.7.1 Type systems review

C++ [Str91] is currently one of the most widely used object-oriented programming languages. C++ types combine the characteristics of interface and implementation types in that they define both the interface and the structure of their objects. Classes in C++ are special cases of types: classes specify properties of first-class objects, while types specify properties of non-first-class objects. C++ inheritance rules are novariant; however, C++ allows polymorphic function and method specifications by using a method (function) signature instead of a name for identification purposes. In the presence of static type checking this is equivalent to a restricted form of static multiple dispatch. Non-first-class objects in C++ are operated upon by free functions; only objects (instances of classes) have methods. C++ also provides parametric types (templates) that can take an arbitrary number of parameters; parametric types can be subtyped.

The C++ type system is not verifiable in general due mostly to its unrestricted use of pointer conversions; however, partial verification is possible and is performed at compilation time. The C++ type system combines interface and implementation inheritance and thus violates the first inheritance requirement. It is not completely uniform as it distinguishes between "data" and "objects" and treats

Table 2.3: Type system expressibility

Type system	Subtyping		Method types	Dispatch	Parametric types				Mutable types	Intersection types
		Inheritance				Bounded	Intra	Inter		
C++	User-defined	Multiple	+	Single[5]	+	-	-	+	+	-
E	User-defined	Multiple	+	Single[5]	+	+	-	+	+	-
O++	User-defined	Multiple	+	Single[5]	-	n/a	n/a	n/a	+	-
O_2	User-defined	?	-	Single	-	n/a	n/a	n/a	+	-
Java 1.1	User-defined	Multiple[3]	-	Single	-	n/a	n/a	n/a	+	-
GJ	User-defined	Multiple[3]	-	Single	+	+	-	+	+	-
JPE	User-defined	Multiple[3]	-	Single	+	+	-	+	+	-
Pizza	User-defined	Multiple[3]	+	Single	+	+	-	+	+	-
JVT	User-defined	Multiple[3]	-	Single	+	+	-/+[11]	+	+	-
ODMG 2.0	User-defined	Multiple[3]	-	Single	-/+[8]	n/a	-/+[11]	-	+	-
DBPL	User-defined	n/a	+	n/a	-	n/a	n/a	n/a	+	-
Modula-90	User-defined	Single	+	Single	-	n/a	n/a	n/a	+	-
Modula-3	User-defined	Single	+	Single	+[9]	-	n/a	n/a	+	-
Oberon-2	Structural	n/a	+	Single	-	n/a	n/a	n/a	+	-
Theta	User-defined	Multiple[3]	+	Single	+	-	-	+	+	-
Ada 95	User-defined	Single	+	Single[6]	+	+	-	+	+	-
Sather	User-defined	Multiple	+	Single	+	+	-	+	+	-
Emerald	Structural	n/a	-	Single	+	-	n/a	n/a	+	Unknown
BETA	Structural	Single	+	Single	+	+	n/a	n/a	+	-
Napier88	-	n/a	+	n/a	+	-	n/a	n/a	+	-
Strongtalk	Structural[1]	Single	+[4]	Single	+	?	-	+	+	-
Cecil	User-defined	Multiple	+	Multiple	+	+	-	+	+	+
BeCecil	User-defined	Multiple	+	Multiple	-	n/a	n/a	n/a	+	+
Mini-Cecil	User-defined	Multiple	+	Multiple	+	+	+	+	+	+
Transframe	User-defined	Multiple	+	Multiple	+	+	+/-[12]	+	+	-
TM	Both	Multiple	-	Single	+	-	-/+[11]	+	-	-
ML	Structural	n/a	+	n/a	+	-	n/a	+	+/-	-
Machiavelli	Structural	n/a	+	n/a	-	n/a	n/a	n/a	+/-	+
Fibonacci	User-defined[2]	Single	Unknown	n/a	+	-	n/a	n/a	+/-	Implicit
ML$_{\leq}$	Both	Multiple	+	Multiple	+	-	+	+/-	+/-	Implicit
PolyTOIL	Structural	n/a	+	Single[7]	+	+	n/a	n/a	+	-
Loop	Structural	n/a	+	Single[7]	-	n/a	n/a	n/a	+	Implicit
TL	Structural	n/a	+	n/a	+	+	-/+[11]	+	+	-
TooL	Structural	Multiple	+	Single[7]	+	+	-	+	+	-
TOPL	User-defined[2]	Multiple[3]	+	Multiple	+	++[10]	-/+[11]	+	-	Implicit

[1] Structural subtyping is the default; however, user can explicitly turn it off when needed
[2] User-defined for *classes*; structural for *algebraic data types*
[3] Only for interfaces; classes have single inheritance
[4] Block types only
[5] Only *virtual* functions are dispatched
[6] Only *tagged types* are dispatched
[7] Dispatch modeled as record field extension/execution
[8] Only system-defined; parameters can only be used for specification of properties
[9] These features are based on *modules* rather than on *types*
[10] The language provides mechanism more expressive then bounded quantification
[11] Always covariant
[12] Only covariant and novariant parameters are allowed

Table 2.4: Type system tests

Type system	*PERSON*	*POINT*	*COMPARABLE*	*STREAMS*	*SORT*	*GENSORT*	*LIST*	*SET*	*APPLY*	*BROWSER*
C++	-	-	-	-	-	+	-	-	-	+/-[7]
E	-	-	-	-	+	+	-	-	-	-
O++	-	-	-	-	-	+	-	-[6]	-	-/+[8]
O_2	-	-	-	-	-	-	-	-	-	??
Java 1.1	-	-	-	-	+	-	-	-	-	+
GJ	-	-	+	-	+	-	-	union	-	+
JPE	-	-	+	-	+	-	-	union	-	+
Pizza	-	-	+	-	+	+	-	union	+	+
JVT	-/+[1]	-	-/+[1]	-/+[1]	+	-	-/+[1]	-/+[1]	-	+
ODMG 2.0	-	-	-	-	-	-	-	union[6]	-	+
DBPL	-	-	-	-	-	+	-	-	-	-
Modula-90	-	-	-	-	-	+	-	-	-	+
Modula-3	-	-	+	-	+	+	-	-	-	+
Oberon-2	-	-	-	-	+	+	-	-	-	+
Theta	-	-	+	-	+	+	+	-	-	+
Ada 95	-	-	+	-	+	+	+	-	+	+
Sather	+	-	+	-	+	+	+	-	-	+
Emerald	+	-	-	+	+	-	-	-	-	+
BETA	-/+[1]	-	-/+[1]	-/+[1]	+	+	-/+[1]	-/+[1]	-/+[1]	-/+[1]
Napier88	-	-	+/-[4]	-	+/-[4]	+/-[4]	+	-	+/-[4]	+
Strongtalk	-	-	+	-	-/+[5]	+	+	union	-/+[5]	+
Cecil	+/-[2]	+/-[2]	-	-	+/-[2]	+/-[2]	+/-[2]	+/-[2]	+/-[2]	+/-[2]
BeCecil	+	+	-	-	+	+	+	-	-	-
Mini-Cecil	+	+	+	+	+	+	+	+	+	-
Transframe	+	+	+	-	+	+	+	union	+	+
TM	-	-	-	-	+	-	+	-[6]	-	-
ML	+	-	-	-	+	+	-	-	+	-
Machiavelli	+	-	-	-	+	+	-	+/-[6]	+	-
Fibonacci	+	-	-	-	+	+	-	+/-[6]	+	-
ML$_{\leq}$	+	+	-	-	+	+	-	union	+	-
PolyTOIL	-/+[3]	-	+	-	+	+	+	-	+	-
Loop	-/+[3]	-	+	-	+	+	+	-	+	-
TooL	-/+[3]	-	+	-	+	+	+	-	+	-
TOPL	+	+	-/+[1]	-	+	+	-	union	+	-

[1] Relies on dynamic type checking
[2] Only suggestive type-checking
[3] While the subtyping relationship required by the test holds, one type can not be derived from the other
[4] If type parameters are explicitly instantiated
[5] This test can be only programmed with blocks rather than messages
[6] Built-in operators for dealing with sets are provided; however, their typing is special (non-user-definable)
[7] If RTTI is present
[8] Even though a type can be examined dynamically, typechecking does not take this into account

attributes in a special way. C++ provides a limited substitutability for object pointers (not for objects). In terms of expressibility, the C++ type system is quite powerful as it has function types, parametric types, orthogonal type constructors, and deals with mutable object types. However, when used on the test suite, the C++ type system fails all tests except for *GENSORT*. Other tests can be programmed, but only with significant type-checking lapses. The reason for this is the fact that C++ does not handle the typing of binary methods since it lacks multiple dispatch. Parametric types in C++ can only be used when fully instantiated, thus limiting a programmer's ability to define polymorphic functions (methods). Type parameters are always unrestricted and novariant. The variant of C++ that includes run-time type information (RTTI) allows for dynamic type checking (allowing it to tentatively pass the *BROWSER* test). Intersection and union types can not be represented by the C++ type system. The C++ type system is also quite complicated.

E [RCS93] is derived from C++ and borrows much of the type and class system from it. Differences between E and C++ lie in the specification of parametric types and in the fact that E is a persistent language. In E (as opposed to C++) parametric types (called *generic classes*) can be specified using a general type parameter restriction mechanism; however, subtyping for parametric classes can not be defined in E. This mechanism allows a programmer to specify restrictions on methods of a parameter type. For example, it is possible to require a certain function (method) parameter type to have a **compare** method with a given specification, thus making the test *SORT* succeed. As a persistent language, E is not completely uniform as its persistence is type-dependent. E also fails the *BROWSER* test.

O++ [AG89] is another persistent language derived from C++. It is similar to C++ in all respects except for provisions for persistence, queries, and a limited form of type-reflection. Persistence in O++ is not type-based, thus its type system is uniform in this respect[10]. Type reflection in O++ is provided by the means of the `is` operator that checks whether a given object has a given

[10] O++ still does not have orthogonal persistence, as its persistence is declaration-based; also, persistent object creation is different from transient object creation.

type. However, the *BROWSER* test in O++ still fails with respect to type checking even though it can potentially be programmed. The *SETS* test also fails as O++ queries do not provide the full power of SQL even though the typing for **forall** and **suchthat** is present in O++ at the operator (non-user-definable) level.

O_2 [LRV92] provides a family of languages, but its type system is the same in all of them, so here the discussion is based upon the CO_2 language, which is based on C. The type system of O_2 makes a distinction between first-class and non-first-class objects (called *values* in O_2; however, values in O_2 are mutable). O_2 uses the term *type* to refer to implementation types of non-first-class objects, and the term *class* to refer to types of first-class objects, which combine properties of interface and implementation types. Thus, the type system of O_2 is not completely uniform in that only first-class objects are manipulated by methods, much like in C++. Inheritance in O_2 is based on implementation subtyping, with an additional ability to add or modify methods. O_2 adopts a covariant signature refinement rule, thus providing for more natural data modeling. However, in the absence of multiple dispatch, this covariant rule results in the loss of static type safety, and thus the type system of O_2 is unsound. O_2 does not provide any kind of parametricity; methods, messages, and types are not objects in O_2. O_2 is uniform in terms of persistence (it is orthogonal to the type). O_2 also makes provisions for dynamic schema evolution; however, this evolution is not type-safe. O_2's type system essentially fails all tests for type system expressiveness, since even the tests that could be programmed would pass type checking and fail at run-time. In [BC95], multimethods are used to provide type safety for covariant specifications in the O_2 programming language. With the addition of the mechanisms described in the article, test *POINT* would succeed, while the others would still fail.

Java [AG96] has recently become a popular language in both academic and industrial communities. Its type system shares many features with that of C++, therefore the discussion will focus primarily on its differences from the latter. The advantages of the Java type system include separation between interface and implementation, better handling of run-time type information, and simplification of the overall type system design resulting in a much more transparent type system. A non-reflexive Java fragment has been shown to be type-safe [DE97], while the full language has certain type deficiencies [Sar97]. On the other hand, the Java type system lacks method types (all methods when considered as objects have the same type in Java 1.1), parametric types (except for statically unsafe parametric arrays, which are built-in), and inherits some of the problematic features of the C++ type system discussed above. Java fails all tests except for *SORT*, the latter being successful due to the presence of *interfaces*. Java fails the *GENSORT* test as its method types are not sufficiently expressive for this test.

Recently, several parametric extensions to the Java type system have been proposed. Generic Java (GJ) [BOSW98c, BOSW98a, BOSW98b] adds parametric types to the static type system, while using the same run-time model (type parameters are erased and do not exist at run-time). Parametric types in GJ are novariant and parameters can only be of reference (class) types. There are also several restrictions on the usage of parametric types and methods, related to the particular type inferencing algorithm used in GJ. GJ passes the test *COMPARABLE* and the union part of the *SET* test in addition to the *SORT* test passed by Java 1.1.

Another parametric type extension of Java is proposed in [MBL97] (JPE). It allows usage of non-reference types as type parameters and also uses *where*-clauses to express requirements on parameter types. In this extension, parametric types are also novariant. The type-checking algorithm is not presented in [MBL97]. This extension has test suite performance identical to that of Generic Java, but provides a more uniform system. These two approaches are informal in that they do not have a theoretical proof of their soundness.

Yet another Java extension is Pizza [OW97] which extends Java with parametric and function types. The approach taken in [OW97] is similar to that of [MBL97], but is much better formalized. In fact, Pizza would have been statically type-safe if not for covariant arrays which were left in Pizza for Java compatibility. It is shown that the resulting type system does not have the subsumption property. However, the same is true of all Java extensions considered so far as well as of Java itself. Pizza passes tests *GENSORT* and *APPLY* in addition to *COMPARABLE* and *SORT* due to the presence of method types. A similar set of extensions is proposed in [AFM97], but without method types and a supporting theory. The latter approach, however, does lift several restrictions placed on the usage of type parameters in Pizza.

A different approach is taken by Thorup [Tho97], where Java is extended with *virtual types* (JVT). Here the choice is made in favor of convenience, and both static typing and (dynamic) substitutability are sacrificed. Because of this, tests *PERSON*, *COMPARABLE*, and *LIST* could be programmed, but would give run-time rather than compile-time errors in incorrect cases. A modification of this approach is presented in [BOW98], which also compares parametric and virtual type approaches.

[BC97] extends Java with multi-methods that are introduced via the so-called *parasitic methods*. The goal of this extension is to add support for multi-methods to the existing language providing full compatibility with Java. [BC97] contains proof of soundness for the resulting system.

The Object Database Management Group has developed a set of standards for object database management systems [CBB+97]. Two of these standards specify an object model (ODMG Object Model) and an object query language (OQL). The object model includes types that specify abstract properties and abstract behavior of their objects. Types are categorized as *interfaces* (abstract behavior only), *classes*, and *literals* (abstract state only). Types are implemented by language-specific *representations*; a single type can have several representations, but only one of them can be used in a single program. Interfaces support multiple inheritance, while classes only support single inheritance (*class extension*). Thus, ODMG Object Model provides separation between interface and implementation. Interfaces in this model can not be instantiated. Abstract properties in ODMG Object Model include abstract state and abstract relationships (two-way mappings). Relationships can only be defined between classes. Abstract behavior is specified as a set of *operations*. Behavior specifications are novariant in both their argument and return types. ODMG Object Model supports single dispatch. Several system-defined parametric container classes are present in the object model, but the user is not allowed to define new ones. Type parameters can only be used in property specifications; operation specifications can not be parameterized. OQL is a strongly typed query language that provides a possibility of explicit dynamically checked type conversions. Set operations in OQL can only be performed on "compatible types" (types that have the least upper bound). Since the notion of the greatest lower bound is not available in OQL, all set operations use the least upper bound for typing purposes. For example, intersecting a set of students with a set of persons would return a result of type "set of persons" even in the case when the type of students is a subtype of the person type. ODMG/OQL fails all tests except for *BROWSER* and the union part of the *SET* test. It should be noted, however, that ODMG/OQL is not a general-purpose database programming language, and therefore its performance on the test suite is not fully indicative of the merits of the ODMG Object Model.

The following group of languages is based, directly or indirectly, on Modula-2 [Wir83]. Modula-2 is neither object-oriented nor a persistent language. Its type checking is verifiable. Modula-2 has a construct for separating interface and implementation in the form of *interface* and *implementation modules*, and the languages based on Modula-2 also use this approach. This mechanism has proven to be very convenient and robust for procedural languages. However, its advantages and disadvantages for object-oriented languages with their primarily type-based approach to both interface and implementation specification are yet to be evaluated.

DBPL [SM94] is one of the persistent languages based on Modula-2. In DBPL, modularity is achieved by using the native language (Modula-2) modularization mechanisms with a special **DATABASE MODULE** construct. In the absence of module persistence, it does not cause problems with orthogonality. Transactions are supported as special procedures. Partial SQL compatibility is provided by the use of a **RELATION OF** type constructor with the appropriate set of operations and first-order constructs **ALL IN**, **SOME IN**, and **FOR EACH**. DBPL also allows updatable and non-updatable views (via **SELECTOR** and **CONSTRUCTOR** procedural specifications). DBPL is not object-oriented; however, it does offer implementation types (with no methods). The DBPL type system is static and non-reflexive. The tests described in Section 2.8 are not applicable to DBPL directly as it is not object-oriented. DBPL would tentatively pass only the test *GENSORT*. The DBPL type system is uniform in that system and user-defined types have the same rights.

Another persistent language based on Modula-2 is Modula-90 [LML+94]. Modula-90 has some rudimentary object-oriented capabilities (single inheritance) and is similar to C++ in that method signatures are novariant, object types are different from other types, implementation and interface hierarchies coincide, and there is only a limited support for function types. Thus, the Modula-90 type system is not uniform. An interesting property of Modula-90 is the presence of type **DYNAMIC**. Data of this type are pairs of values and their types expressed as values of a compound type **DATATYPE**. Type **DATATYPE** is the type of the representations of all data types in the system; it can be used independently of **DYNAMIC** type to store, retrieve, and operate upon various data types. Any value can be coerced to the type **DYNAMIC**. Modula-90 provides a special operator **TYPECASE** that provides a type-safe interface to the values of type **DYNAMIC**. The language also provides a set of predefined type operators that can be used to operate on values of type **DATATYPE**. Thus, Modula-90 provides type-safe immutable type reflection. The Modula-90 type system is static and non-parametric; however, it does provide orthogonal persistence. The only expressibility tests that would succeed in Modula-90 are *GENSORT* and *BROWSER*. Modula-90 provides incremental type checking, including its dynamic variety.

Modula-3 [Har92] is another language with object-oriented extensions based on Modula-2. Modula-3 is not a persistent programming language and its object extensions are similar to those of Modula-90; it also has **TYPECASE** statement that gives a programmer the ability to request dynamic type checking in a type-safe manner. Modula-3 has parameterized *modules*; however, parameterized *types* are not allowed. Modula-3 passes the tests *COMPARABLE*, *SORT*, *GENSORT*, and *BROWSER*.

Another descendant of Modula-2 is Oberon-2 [MW93]. Subtyping in Oberon-2 is based on *record extension*, also known as *structural subtyping*. The subtyping relationship is predefined for atomic types. Thus, Oberon-2 has multiple subtyping. Oberon-2 supports single dispatch. Subtyping of procedure (method) types is based on novariance of argument and result types; only the receiver type is covariant. Parametricity is not supported by Oberon-2. Separation between interface and implementation is supported at the level of *modules* in the same manner as in all Modula-2 based languages. Oberon-2 also has an extended **WITH** statement that allows a programmer to dynamically inspect the type of an object and act according to it. This statement is similar to the **TYPECASE** statement in Modula-3 and Modula-90 discussed earlier. The type system of Oberon-2 would pass the tests *SORT*, *GENSORT*, and *BROWSER*.

There has been a proposal for adding parametric types and methods to Oberon [RS97]. In this proposal, parametric types are novariant, and the type checker ensures that a parameter type substitution exists that satisfies the rules for matching arguments of a particular call. Unfortunately, neither the typechecking algorithm nor the proof of its soundness are present in [RS97].

Lagoona [Fra97] is another descendant of Oberon. It focuses on *messages*, *objects*, and message passing. In contrast to most object-oriented languages, a message in Lagoona is an independent entity. An interface (*category* in Lagoona) is a set of messages. Each message has a type specification. There can be several *methods* implementing a message, each for a different receiver type. Implementation in Lagoona is specified by a *class* (usually a record type); when a *category* meets a *class*, a *type* (combination of interface and implementation) is born. No code inheritance is possible in Lagoona, only structure inheritance is possible. Variables in Lagoona can be specified using either *category* or *class*. Lagoona thus provides a complete separation between interface, implementation (representation), and code (methods). When a message is sent to an object, an object can forward (**resend**) it to another object. Other features of the Lagoona type system include single dispatch and single inheritance. Unfortunately, the article [Fra97] provides insufficient information to test the performance of Lagoona on the test suite.

The language Theta [DGLM95] combines the expressive power of parametric and inclusion polymorphisms. It also provides *where*-clauses that give its type system a flexibility similar to that provided by matching. Theta has single dispatch, multiple inheritance for types (interfaces), and single inheritance for classes (implementations). Theta only allows novariant parametric types. Method types can be specified in Theta and methods (functions) can be used as first-class values. Theta also has reflexive capabilities in the form of the **typecase** operator. Theta's type system is verifiable, uniform, and static. It is also quite expressive. However, the its soundness has not been proven. Theta passes the tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER*.

The language Ada 95 [Ada95] is a procedural language with some object-oriented features. Ada 95 has a remarkably strong support for parametricity: *generic packages* (parametric modules) and *generic subprograms* (parametric functions) can not only be instantiated, but also used as parameters of other generic entities, thus providing for great flexibility. Parameter types can be constrained by a **with** clause requiring the type to have a method with a particular signature, which can also be parameterized. However, Ada generics have to be fully and explicitly instantiated before they can be used. Another interesting and powerful feature of Ada is the notion of an *access type* which generalizes such notions as "pointer" and "reference", allowing also for user-defined access types. Ada also has function types, and function arguments can be specified as **in**, **out**, or **inout** according to the role they play. Ada uses its *package* mechanism (similar to the module mechanism of Modula-2) to provide separation between interface and implementation. On the other hand, object-oriented Ada 95 features appear to be relatively weak: in order to be used for dynamic dispatch, a type has to be explicitly declared as **tagged**. In order to be able to use subtyping (inclusion) polymorphism, a programmer has to use a special form of parameter specification. Ada provides single dispatch and single inheritance with novariant methods. Subclassing in Ada is limited to record extension. In Ada, types can be examined dynamically; data of a type can be converted to any other type, and such a conversion is dynamically type-checked. The Ada type system is very complicated and it places emphasis on *static* rather than *dynamic* polymorphism. Ada will pass the tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER*, provided generic packages rather than types are used.

Next, the object-oriented language Sather [SO96] will be considered. Sather is based on the much better known Eiffel [Mey88]; however, Eiffel is not discussed here as one of the primary goals behind the design of Sather was to remedy typing problems present in Eiffel.

Sather has multiple interface and implementation inheritance hierarchies almost independent of each other (concrete implementation types called *classes* in Sather must have leaf interface types). Implementation subtyping in Sather corresponds to textual inclusion with replacements. Sather uses single dispatch and is strongly and statically typed. Therefore its methods are covariant on the receiver and contravariant on other arguments. Sather also provides partial closures of its methods and iterators as first-class values. It has parametric implementation types that can use a form of bounded polymorphism to constrain type parameters. These parametric types are novariant and they can not be used until fully instantiated. Method arguments in Sather can be specified as **in**, **out**, or **inout**, according to the role they play. The Sather type system correctly handles all these cases. Argument types and local variables can be specified by using either Sather types or Sather classes. In the former case, the parameter class has to be a subtype of the given type; in the latter case, the parameter class has to match the specification exactly. The language also provides a novel notion of *iters* (iterators) that are an object-oriented generalization of loop control structures. Sather also provides a special compound type **TYPE** and operators **typeof** and **typecase**. Thus, it has type-safe immutable type reflection. The type system of Sather therefore possesses verifiability (even though it has not been formally proven) and satisfies inheritance and (partially) uniformity and expressibility requirements. It passes tests *PERSON*, *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *BROWSER* and fails on tests *POINT*, *STREAMS*, *SET*, and *APPLY*. Note that *APPLY* fails because parametric types in Sather require full instantiation before they can be used.

Emerald [RTL$^+$91] is a non-traditional object-oriented language that combines features of class-based and delegation-based object-oriented languages. Objects are created in Emerald by a special syntactic form called *an object constructor*. The object constructor plays a triple role: first, it defines the object's implementation; second, it defines a publicly visible object interface (*type* in Emerald); third, it denotes the process of object creation itself. Subtyping in Emerald is structural, as a type is understood as a set of signatures. Types are also objects that can be created by object constructors; thus, a user can define new types, including parametric ones. The type checker uses user-defined types along with system-supplied ones to typecheck a program statically. Since types are objects, dynamic type checking is also possible. The Emerald type system is therefore verifiable, uniform, reflexive, satisfies inheritance requirements, and is quite expressive. However, its soundness is unknown. Emerald passes the tests *PERSON*, *STREAMS*, *SORT*, and *BROWSER*.

BETA [MMPN93] is a unique object-oriented language that unifies the notions of class, object, and procedure (method) via its notion of a *pattern*. Patterns can contain other patterns (such as member objects, code fragments, and types). BETA also has *fragments* that play the role of modules and can also be used to separate interface from implementation. BETA fragments can be regarded as high-level restricted patterns since they operate on the same basic principles. Patterns can be *virtual*; a virtual pattern can be extended by adding code fragments in places specified by the **inner** placeholder (a.k.a. method extension as in Simula-67 [BDMN79]), by extending a virtual pattern with additional members (a.k.a. record extension), or by supplying a class pattern in place of a virtual one (a.k.a. parametric instantiation). Virtual patterns have to be explicitly declared as such. Unification of all language concepts using the notion of pattern makes it possible to design a very powerful language based on only a couple of orthogonal principles. While the language design simplicity is very impressive, the resulting language is quite unconventional. Structural subtyping in conjunction with a class substitution mechanism makes the type system of BETA statically unsound; dynamic type checks are inserted to ensure type safety. Due to the uniformity of BETA, all tests except for *POINT* can be coded in it, but only *SORT* and *GENSORT* have static type safety. Illegal usage in other tests would cause run-time type errors.

In the persistent object-oriented language VML [KT92], all objects that are instances of object types (called *classes*) are persistent while all (non-first-class) objects of non-object types (called *data types*) are transient. However, a value of a non-object type can become persistent if it is referenced from a persistent object. Thus, VML does not have orthogonal persistence. VML object types are first-class objects and as such belong to their respective *metaclasses*. Metaclasses define some of the essential class methods, for example the methods for the *inheritanceBehavior* message that is used when a method for a message is not found in the receiver class. Thus, VML allows user-defined method inheritance due to its "classes are objects" concept. However, VML data types are not first-class objects and thus VML only partially satisfies the type reflexivity requirement. Inheritance in VML can be tailored to specific application needs by using the user-definable *inheritanceBehavior* message found in an appropriate metaclass. VML does not have a verifiable type system, therefore the tests are not applicable to it.

The persistent programming language Napier88 (version 2) [MBC+96] is not an object-oriented programming language; however, its type system is quite powerful. In Napier88, parametric types can be built freely from the basic types, type constructors, and type variables. Parametric procedures (procedures with parametric types) can also be defined and are first-class objects. Napier88 types are purely implementation types; however, Napier88 provides the type constructor **abstract** that may be considered as an interface type as it provides existential quantification over witness type(s). Thus, the type system of Napier88 satisfies all expressibility requirements not related to the notions of subtyping and inheritance. However, since Napier88 does not have the notions of subtyping and inheritance, it fails the requirements related to those notions. In Napier88, parametric types can not be used until fully instantiated. Napier88 provides support for a limited form of linguistic reflection via dynamic *environments*. Environments are dynamically typed structures that provide bindings of names to Napier88 entities. Environments can be dynamically modified, inspected and used in expressions. Environments can be persistent.

Napier88 has a special type **any** that is a union type of all types in the system. A special **project** operator can be used to deal with values of type **any** in a type-safe manner. This operator is similar to **TYPECASE** of Modula-90. Since any value can be injected into type **any**, the above mechanism makes run-time data type inspection possible. Thus, the type system of Napier88 has type-safe type reflexive capabilities.

The persistent store of Napier88 is an object of type **any** that holds a typed collection of objects. The objects from the persistent store can be projected from **any** and operated upon transparently after that. Napier88 uses a persistence model based on reachability from the root (persistent store) object. Thus, the persistence in Napier88 is orthogonal to type.

In spite of the power of its type system, Napier88 would only unconditionally pass the test *LIST*. It would, however, tentatively pass the tests *COMPARABLE*, *SORT*, *GENSORT*, and *APPLY* if usage of explicit type parameters in parametric calls were allowed. In other words, the burden of inferring correct parametric instantiation from the code in Napier88 is placed on a programmer rather than on the type checker.

Next, purely object-oriented programming languages' type systems will be reviewed. The first language to be considered in this group is Strongtalk [BG93] which is a statically typed version of Smalltalk [GR89]. Smalltalk is widely regarded as the first purely object-oriented language; however, it has no static type checking.

In Strongtalk, everything (including types, called *classes* in Strongtalk, methods, and messages) is a first-class object. Thus, Strongtalk is uniform[11]. All Strongtalk objects can be operated upon and modified at run-time and therefore Strongtalk is reflexive (the reflexivity is type-unsafe). Strongtalk separates interfaces (*protocols*) from implementations (*classes*), provides subtyping and matching. It also has parametric (novariant) types and messages, block types, and union types. However, the user has to explicitly specify a mechanism to guide parametric type inferencing, resulting in the loss of substitutability. It is also unclear from [BG93] whether Strongtalk allows bounded parametric types. Strongtalk provides single inheritance and single dispatch. Subtyping is structural, but the user can use *brands* to restrict it. Even though Strongtalk was not designed to be a persistent language, it does provide uniform persistence of its objects in a so-called *image*. Overall, the Strongtalk type system is quite expressive, uniform, and reflective. It passes the tests *COMPARABLE*, *LIST*, *BROWSER*, and, possibly, *GENSORT* (if parametric type bounds are allowed). The union part of the *SET* test is also passed. Strongtalk would fail the tests *PERSON*, *POINT*, *STREAMS*, as well as *APPLY* and *SORT* (the latter two could be programmed, but only with blocks rather than messages).

Cecil [Cha93, Cha92b] is a delegation-based language that has both implementation and interface types (the former are called *representations* while the latter are called *types*). Types in Cecil are used for suggestive type-checking only as Cecil's multiple dispatch is done according to representations. Cecil's type checking is suggestive because it might report false errors or miss real ones, therefore its type discipline is not enforceable. Cecil's does not support incremental type checking. Cecil uses multiple dispatch and provides covariant specification of specialized arguments together with contravariant specification of unspecialized ones. Closures and methods are first-class objects in Cecil; they are contravariantly typed. Cecil also provides novariant parametric types and methods as well as type parameter bounds. Parametric types in Cecil can be instantiated either explicitly or implicitly; in the latter case, the user has to provide a hint to the type inferencing algorithm. This behavior results in loss of substitutability. Cecil has multiple inheritance, union and intersection types. Overall, the Cecil type system is quite expressive and uniform, while being non-reflexive[12] and static. It also satisfies the inheritance requirements. Since Cecil type checking is only suggestive, it is difficult to apply the tests to it. However, tests that Cecil would tentatively pass include *PERSON*, *POINT*, *SORT*, *GENSORT*, *LIST*, *SET*, *APPLY*, and *BROWSER*. It would fail the tests *COMPARABLE* (due to the restrictions on parametric type bounds) and *STREAMS* (due to the novariance of parametric types).

There are several extensions/modifications to the original Cecil language. One of these extensions is BeCecil [CL96, CL97]. BeCecil is a statically typechecked version of Cecil. It supports block and modular structure, has extensible objects and an extensible type system, and its type system has been formalized. However, soundness and substitutability properties have yet to be proven. BeCecil also has a novel notion of *acceptors* which can be considered as an object-oriented generalization of the assignment operator. However, BeCecil does not have parametric types while sharing most of the other features with Cecil. The absence of parametricity contributes to the decreased expressibility of the BeCecil type system as compared to that of Cecil. BeCecil passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, and *LIST*, and fails the rest.

Another modification (Mini-Cecil) is described in [Lit98]. Mini-Cecil strives to achieve a combination of static typing and a very general form of parametric polymorphism. Mini-Cecil also has *frameworks* which are basically interfaces with what appears to be an analog of **selftype**. The frameworks can be used to separate interface and implementation, as well as to achieve the effect of matching. Methods in Mini-Cecil are first-class objects; multimethods and multiple inheritance are supported. Subtype clauses can have a form of **forall** $\overline{\alpha}$ C_1 **isa** C_2: C_3 **isa** C_4, where $\overline{\alpha}$ is a set of free type variables and C_i are type specifications. The resulting type system is very expressive;

[11]Except for a possible uniformity breach in the form of direct attribute access.

[12]Being a delegation-based language, Cecil has language reflection; however, the type system of Cecil is not reflexive as Cecil types are not objects and can not be manipulated by the language. More precisely, *representations* of Cecil are reflexive while *types* are not.

it can even be argued that it is *the most* expressive type system possible. However, the typechecking seems to be undecidable. The algorithm proposed is a conservative approximation, and its soundness is yet to be proven. It is also unknown if the resulting type system has substitutability. Mini-Cecil would pass all tests except for *BROWSER* as it does not have reflexive capabilities.

Transframe [Sha97] allows the user to specify whether a parameter of a parametric type is to be covariant or novariant (*type-exact*) and to constrain it by giving it an upper bound. Subtyping and subclassing (interface and implementation inheritance) are different concepts in Transframe. There is a distinct name **selfclass** that allows classes to support matching. The language unifies the notions of *class* and *function* (like BETA). Transframe also supports multiple dispatch. There are provisions for dynamic type checking and dynamic schema evolution. Transframe implicitly instantiates parameter types in expressions; unfortunately, there is no formal proof of type safety. In fact, it can be shown that the type system presented in [Sha97] is *not* type-safe. Overall, the Transframe type system is verifiable, very expressive, almost uniform, and dynamically reflective. However, it is unsound. Transframe would pass all expressibility tests except for the test *STREAMS* (due to its inability to represent contravariant type parameters) and the intersection part of the test *SET* (due to the absence of intersection types).

CLOS (Common LISP Object System) [BDG+88] is a reflexive language, with all the power of Common LISP reflection. CLOS has types and object types (called *classes*), the latter being a subset of the former. CLOS types are implementation types; they do not specify any interface. However, CLOS classes combine implementation and interface definitions. Since CLOS makes a distinction between object and non-object types (where only object types define interfaces and are subject to inheritance), the CLOS type system is not completely uniform. CLOS classes are objects that belong to metaclasses, which are also objects; CLOS messages, methods, and functions are also CLOS objects that can be operated upon and changed at run-time, so CLOS is fully reflexive and dynamic. Subclassing in CLOS is slot collection extension with slot types changed covariantly (a type of a slot is the intersection of the types specified for this slot in all of the class' superclasses). Messages (called *generic functions*) are also covariant. CLOS is not statically typed. A CLOS message can be dispatched to yield an appropriate method (or a combination of methods) according to the class or value of all arguments (multiple dispatch). Methods are covariant on all arguments[13] since CLOS has multiple dispatch. If more than one method is appropriate for the given arguments, a user-definable way of constructing the function to be executed out of *all* appropriate methods is employed. In the body of a method, a special function **call-next-method** can be called to invoke the next applicable method. This capability is analogous to the **inner** construct of Simula-67 [BDMN79] and is much more powerful. Overall, the dispatch mechanism of CLOS is the most powerful of all known mechanisms, if the consideration is limited to classes and values. CLOS can not dispatch on types. Updates in CLOS are invoked on slots directly or by using an appropriate message. CLOS is not statically type checked, therefore a run-time error is signalled if a value assigned to a slot does not conform to the slot's type specification. Since CLOS is not statically typed, the tests are in general not applicable to it; however, CLOS would pass *POINT*, *APPLY*, and *BROWSER* tests if its type discipline were enforceable.

Dylan [App94] is an imperative programming language similar to CLOS. While there are certain differences between the two, they are almost identical in terms of their type systems. Dylan has more control over the defined classes, as Dylan classes can be sealed (only subclassable by the library where they belong) or open, primary (there is only single inheritance of primary classes) or free, abstract (all superclasses of an abstract class must be abstract) or concrete. There is also support for singleton types, but not singleton classes. Multiple dispatch in Dylan is also different from that in CLOS in that in Dylan all arguments are equal, and the method specificity is defined by a class precedence list. Dylan also has modules with import and export lists and module libraries.

TM [BBdB+93] is an object-oriented persistent language with many functional features. TM has a verifiable type system based on [Car88]. However, the soundness proof seems to be missing. TM's type hierarchy includes user-definable sorts (atomic, immutable types) and classes. Sorts and classes have representation (implementation) types that are almost hidden inside of them. Methods and

[13] More precisely, methods are covariant on those arguments that are constrained by class specifications.

types are not first-class values in TM. TM method specification uses **selftype** to achieve the effect of matching. Since this is the only polymorphic mechanism in TM, specifications are covariant and substitutability does not hold, as functional updates are present. No method redefinition mechanisms are provided in TM. The TM type system extends the type system of Cardelli [Car88] with a powerset type constructor. Since TM is stateless, powerset types are covariant. TM allows enumerated as well as predicative sets as primitive language expressions. Enumerated sets in TM can only be homogeneous, while predicative sets can be heterogeneous up to subtyping. Predicative sets in the presence of the record-based type system of [Car88] and the absence of updates play a role of embedded queries that are highly integrated with the rest of the language. TM provides several levels of constraint specification mechanisms which are also set-based and resemble relational constraint systems. TM also provides first-order set operations. However, the set operations require special treatment and are not messages in the usual object-oriented sense. It is also unclear how well different type constraints for different kinds of sets interact with each other. TM has modules that define their persistent components by names, and everything those objects refer to is also implicitly persistent. Thus, TM provides a combination of static name-based and dynamic reachability-based persistence, completely orthogonal to the type. Overall, the TM type system is verifiable, sound, supports both interface and implementation inheritance (though they are not completely independent of each other), is almost uniform, partially expressive, and provides support for declarative queries. However, it does not support substitutability and is non-reflexive and static. It would unconditionally pass tests *SORT* and *LIST*. Test *SET* also succeeds because of built-in support for set operations. However, it would not be possible to construct user-defined types with the same functionality.

Most of the following languages borrow much of their expressiveness from ML [MTH90]. ML is a language with both functional and imperative flavors; it also has some object-oriented features. This can be said about almost all the languages discussed below.

Standard ML [MTH90] is a functional language with some imperative features. It is strongly typed and provides provably decidable and sound type checking. In the ML type system, all type information is *inferred* by the type checker. Addition of explicit type annotations and declarations is considered in [OL96]. Standard ML also provides a very sophisticated module system, where each module (*structure*) has its type (*signature*). However, ML's structures are more like abstract (implementation) types than modules, as they are designed to shield their internals from the rest of the program and not to handle separate compilation or similar tasks. There is no notion of subtyping in Standard ML, except for signature matching, which can be considered as restricted structural subtyping for abstract (interface) types. Highly parametric types are supported in Standard ML. They can arbitrarily include type variables and can be user-defined. Thus, the type system of Standard ML is uniform. For interface types (signatures) there are also functions that map signatures to signatures (functors). Function types in ML can also be polymorphic. Polymorphism in Standard ML function types is expressed via type expressions that have to be "pattern-matched". For example, the type of the identity function in Standard ML is `'a -> 'a`, where `'a` is a type variable. This kind of polymorphism is uniform in that it allows user-defined parametric types. Standard ML, being in essence functional, allows updates on so-called **ref** types. These types are reference types somewhat similar to pointer types in C or C++. **ref** types have a restricted parametricity, as an argument of a **ref** type must always be a monotype (e.g. see discussion in [Wri93]). Types are not objects in Standard ML even though they can be operated upon by functions similar to those that operate on ordinary values. Overall, the type system of Standard ML is theoretically sound, very expressive and uniform, while lacking inheritance and being only partially reflexive. This type system would tentatively[14] pass tests *PERSON*, and *SORT*, *GENSORT*, and *APPLY* (for the test *SORT* use of modules rather than types is required).

Another language from this family is Machiavelli [OBBT89, BO96] which is a persistent language that extends ML by adding more polymorphism as well as query and view support. Machiavelli has a verifiable and provably sound type system. It adds record inclusion polymorphism to ML by using type variables of the form `(a'')`, that correspond to an arbitrary record extension. Thus Machiavelli allows "more polymorphic" types than ML. Machiavelli is also able to automatically

[14] Provided that the notion of subtyping is substituted by the notion of code reuse.

maintain more sophisticated (even non-covariant) type constraints on *description types*[15]. It is therefore possible for Machiavelli's type checker to statically infer an error in case a join of two sets of records whose types do not have a greatest lower bound is attempted. Machiavelli's type system treats description and other types differently. Thus, it is not completely uniform. In Machiavelli, a special set type constructor {} is introduced and query operations are defined for objects of set types. Machiavelli extends the type system of ML to be able to deal consistently with type inference of generalized relational operations. Machiavelli also provides views similar to those in relational databases. Namely, a Machiavelli view is defined as a function that returns a projection of a given set over some appropriate type[16]. Overall, the type system of Machiavelli is verifiable, sound, very expressive, reflexive, partially uniform, partially dynamic, and capable of supporting views. However, it lacks interface inheritance. It passes the same tests as ML with the addition of the *SET* test due to the special support of sets by the language and the type system.

Fibonacci [AGO95] is a persistent object-oriented language that is a descendant of Galileo [ACO85] which is, in turn, based on ML. It has some functional and imperative features and possesses a verifiable, provably sound type system; it also has multiple inheritance and single dispatch. In Fibonacci, *object types* are independent from each other in terms of subtyping; however, *role types* form independent directed acyclic graph (DAG) subhierarchies for each *object type*. For example, an object of type **PersonObject** can play roles **Person**, **Employee**, **Student**, and **TeachingAssistant** (that is both **Employee** and **Student**). Fibonacci roles can be dynamically created and Fibonacci objects can dynamically acquire new roles. Fibonacci types are not objects in the language. Method arguments in Fibonacci are contravariant, while their results are covariant. Fibonacci supports a distinction between methods and functions: methods are attached to role types and are not Fibonacci objects, while functions are independent and are first-class values. Fibonacci also has non-object and non-role types, such as basic types, class types, function types, and association types. These types form a hierarchy independent of that of object and role types. Fibonacci defines some built-in parametric types (**Class**, **Sequence**, and **Association**). However, it is unclear from [AGO95] if the user can create new parametric types. In Fibonacci, objects of the same object type can have different implementations. These implementations are defined at the time of object creation. Thus, an implementation in Fibonacci is not a part of an object or role type specification. Updates in Fibonacci are allowed on special novariant **Var** types. In Fibonacci, the syntax of message sends determines the strategy of the method lookup. There are two strategies: *upward lookup*, that corresponds to the standard lookup procedure in the presence of multiple inheritance, and *double lookup*, that first tries to find an appropriate method in the subroles of the role it has started from. Fibonacci offers declarative query operators on parametric **Sequence** types. These are types of immutable sequences that are supertypes of their respective mutable **Class** and **Association** types. Thus, the query facilities of Fibonacci are also applicable to Fibonacci classes and associations. Fibonacci has a reachability-based orthogonal persistence model. Everything accessible from the top-level environment automatically persists between sessions. Thus, Fibonacci persistence is orthogonal to both interface and implementation types. Overall, the Fibonacci type system is verifiable, provably sound, provides different inheritance for interface and implementation types, and has inclusion polymorphism (substitutability). It is also expressive, almost uniform, and is capable of supporting query typing. However, it is non-reflexive and static. The type system of Fibonacci would pass the same tests as that of Machiavelli.

The language $ML_{\leq}$ [BM96b, BM96a] is an extension of ML with subtyping and higher-order polymorphic multi-methods. It has type inference, strong static type checking, and substitutability. ML-like *type constructors* provide parametric polymorphism. Type constructors in $ML_{\leq}$ can be specified as covariant, novariant, or contravariant. The formalism used is based on the systems of type constraints. In this theoretical language, no separation between interface and implementation is provided. Handling of imperative (mutable) types is borrowed from ML and is quite restrictive w.r.t. polymorphism. Another restriction placed on $ML_{\leq}$'s type system is the requirement that all types that have a subtyping relationship should have the same number of arguments as well as the same variance specification for them. Thus, this type system fails the test *STREAMS* as it requires

[15] Description types are ML types that do not include $\rightarrow$ outside of **ref**.

[16] It is unclear whether Machiavelli handles view updates.

a subtyping relationship to be established between parametric types of different variances. Overall, the $ML_{\leq}$ type system is expressive, verifiable, sound, and static. It passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, *APPLY*, and the union part of the *SET* test. However, the inability of the system presented in [BM96b] to deal with recursive constraints makes generalization of the system to unrestricted subtyping of parametric types quite difficult.

There are several other approaches to adding subtyping to ML, but none of them deals with multi-methods. Aiken and Wimmers [AWL94, AW93] proposed a system that finds a solution for a system of subtyping constraints; this system can deal with recursive constraints. Pottier [Pot98] also proposed a system in which recursive constraints are allowed; instead of solving the constraints, his system proves their consistency (like that of $ML_{\leq}$). [Seq98] also adds subtyping and user-defined type constructors, as well as constrained types, to an ML-style type system. The resulting system appears to be similar to that of $ML_{\leq}$, but lacks its ability to deal with multi-methods. Complexity results related to solving subtyping systems appear in [Reh98].

[MW97] proposes a type system for a core subset of the purely functional language Erlang [AVWW96]. The type system is similar to the one proposed by Aiken and Wimmers [AWL94, AW93] and uses constrained type entailment for type verification. The main difference is the absence of function types, general unions, and intersections. The system is provably sound and presumably complete. Addition of function types to the system makes it incomplete, and the proof of soundness in this case is absent. [MW97] includes decidable algorithms for type inferencing, signature verification, and constraint simplification.

Castagna, Ghelli, and Longo [CGL95] proposed a variant of λ-calculus ($\lambda\&$-calculus) dealing specifically with multi-methods. Several important results (generalized subject reduction, Church-Rosser etc) are proven. It is also shown how the calculus can be used to model inheritance, matching, and multiple dispatch.

[CO94] presents a type system where a full-fledged support for mutable types is added to an ML-style type system. The approach taken by the authors separates mutable and immutable types by creating a parallel language syntax. In essence, every functional language construct has its imperative counterpart. Interaction between mutable (imperative) and immutable (functional) language components is restricted by a set of type validity rules that restrict polymorphism of data types for the data passed between the two language components. The approach of [CO94] appears to be sound, but soundness seems to be achieved at the expense of language simplicity and type system transparency.

Next, theoretical programming languages and systems that are designed specifically for object-oriented programming will be reviewed.

PolyTOIL [BSG95, BSG94] has a verifiable and sound type system and single subtyping. PolyTOIL identifies subtyping with substitutability, while providing a concept of *matching* (subtyping up to **MyType**). The latter is introduced to allow for covariant method specification. Subtyping in PolyTOIL is structural, as is matching. The language allows for both subtyping and matching constraints. Matching is used as a mechanism of specifying constraints that are weaker than substitutability and is therefore useful for updatable types. In [Bru96] it is suggested that subtyping should be dropped altogether as matching is more intuitive and more flexible. In [BSG95], there are distinct notions of *object types* and *class types*. The former are interface types, while the latter are implementation types. Namely, object types specify signatures (type information) for methods applicable to the objects of this object type, while class types specify instance variables and code for methods applicable to objects that belong to the class. Classes are used to create new objects. They are produced by applying functions whose arguments are values used to initialize the produced objects as well as the argument types for parametric classes to their arguments. Classes can use inheritance with redefinitions. Parametric types in PolyTOIL are pattern functions of their parameter types. A notion of a function type is also present in PolyTOIL. However, it is only used in specifications and during type-checking and is inherently different from either class or object type. Overall, the type system of PolyTOIL is verifiable, sound, expressive, almost uniform, and satisfies inheritance requirements. However, it is static and non-reflexive. It passes tests *COMPARABLE*, *SORT*, *GENSORT*, *LIST*, and *APPLY* and fails the rest. The *PERSON* test fails, because while record subtyping allows method type redefinition, record extension does not.

Loop [ESTZ95, EST95b, EST95a] is a theoretical language similar to PolyTOIL. There are no explicit type annotations in Loop and the typing is inferred automatically. Loop has a concept of *subclassing* where one class *inherits* from several other classes. Subtyping and subclassing in Loop are different concepts. Loop enjoys provably sound type-checking and a state semantics given by its translation to Soop. Function types are present in Loop; however, in the absence of explicit type annotations, they are only used internally for type checking purposes. Loop classes are mechanisms for creating objects. Subclasses do not necessarily correspond to subtypes and class and type inheritance hierarchies are different. The type hierarchy in Loop is implicit, while the class hierarchy is explicitly specified by the programmer. Subclassing can be multiple and both methods and instance variables can be added, inherited, or modified. Since Loop is a "theoretical" language, it does not have any syntactic sugar for subclassing which makes Loop inheritance rather difficult to use. Updates in Loop are allowed for instance variables only. The semantics of these updates is given by their translation to Soop. Loop does not have parametric types. Subtyping in a similar system has been shown to be decidable in [TS96]. The type system of Loop is verifiable, sound, partially expressive, almost uniform, partially reflexive, static, and satisfies the inheritance requirements. It passes the same tests as PolyTOIL. The type checking mechanism uses constrained types. The system does not attempt to find a solution to the system of constraints it generates; rather, it verifies that such a system is non-contradictory. The theory guarantees that in this case the system has a solution and the program is considered to be type-correct.

TL (Tycoon Language) [MS92, MMS94] is based on the $F_{<:}$ system [CMMS91, Car93]. From $F_{<:}$ it borrows constrained (bounded) parametric types and type operators, as well as polymorphic functions and partial type inference. It is also uniform in its treatment of functions (including higher-order ones) and atomic values. In addition to these features, TL has mutable types, modules, and **typecase** statement. Even though TL is designed on the basis of a formal system ($F_{<:}$), its type system features have not been mathematically proven. Thus, the questions of soundness and decidability remain open for the TL type system. TL is an orthogonally persistent programming language. Since TL is not an object-oriented language, the tests are not applicable to it.

In the TooL language [GM96], an attempt is made to combine the notions of subtyping, matching, and bounded universal quantification. The resulting language is quite powerful in terms of supporting different kinds of relationships between types. However, it has significant complexity and requires good anticipation by type specifiers to correctly choose the kind of type relationship that is needed *before* any subclasses of the class in question are created. The theoretical aspects of the language do not seem to be fully developed, as neither soundness nor decidability of type checking has been proven. TooL supports single dispatch. Interface and implementation subtyping (termed respectively as subtyping and subclassing) are different in TooL. Parametric types can be specified, and the type parameters can be bounded. Information presented in [GM96] is insufficient to judge the uniformity and reflexivity of the type system; however, it seems to be static and very expressive. [GM96] also states that TooL is a persistent language; however, nothing else is said about its persistence. The performance of this type system with respect to the test suite is identical to that of the type systems of PolyTOIL and Loop.

The language TOFL [QKB96] is a theoretical object-oriented functional language. It has multiple dispatch, novariant argument redefinition, function types, and parametric types. TOFL allows subtype specifications of the form "if x is a subtype of Eq, then $list(x)$ is a subtype of Eq for any type x". All parametric types in TOFL are covariant, except for functionals which are novariant in their first argument (function argument position). This is justified since TOFL is a functional (stateless) language. TOFL has a verifiable and provably sound type system. The TOFL type system is also quite expressive as it passes the tests *PERSON*, *POINT*, *SORT*, *GENSORT*, *APPLY*, and the union part of the test *SET*.

2.8 Conclusions

In this chapter, the set of requirements for a type system of an object-oriented database programming language was formulated. In order to be able to assess expressibility of various type systems, a set of tests was developed. A large number of existing type systems and languages was reviewed and assessed according to their ability to satisfy the requirements and handle the test cases.

Of the languages reviewed, none was found to completely satisfy the requirements laid down in Section 2.8. None of the provably sound type systems has passed the majority of the tests. However, every test was passed by at least one type system thus showing that the necessary mechanisms have already been developed. It is their consistent and theoretically sound combination that remains elusive so far.

The type system presented in the next chapters is the result of my attempt to produce such a combination. It was a challenging task, and its solution will allow us to the develop a theoretically sound, reflexive, uniform, and dynamic persistent object-oriented programming language.

Chapter 3

The Type System Design

This chapter gives an informal description of the type system. It illustrates the main ideas behind the developed type system, gives examples of its use, and discusses design decisions. The formal treatment of the type system is given in Chapter 5.

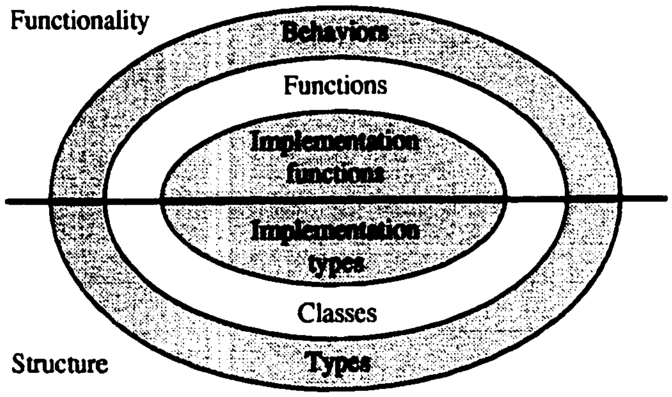

Figure 3.1: Type system layers

The type system consists of three specification layers. Each layer consists of two components: one deals with functionality (dynamic component), and the other with structure (static component). This taxonomy is depicted in Figure 3.1.

The most abstract layer consists of *types* (abstract data) and *behaviors* (abstract functionality). This is the interface layer that has no concrete description of object structure and functionality. This layer defines the object interface only and is primarily used for typechecking purposes. It is described in Section 3.1.

The intermediate layer consists of *classes* (high-level structure) and *functions* (high-level code). It is this layer that defines most of the code to be executed when a program runs. *Classes* are also used for object creation and extent maintenance. This layer is described in Section 3.2.

The lowest layer allows the programmer to use a low-level language to specify data and functionality. This is the layer at which many system primitives are defined. An ordinary programmer usually does not need to know or care about the existence of this layer unless heavy optimization or interoperability issues are involved. This layer consists of *implementation types* and *implementation functions* that will be considered in Section 3.3.

The three-layer design is useful in shielding the user from the internal complexity of the system. Namely, an ordinary user posing queries against a database would only have to be familiar with the outermost layer (types and behaviors). An application programmer needs to know about the second abstraction layer (functions and classes) in addition to the first one. Only a database administrator, implementor and designer should operate at all the three layers, using the implementation layer for low-level optimization and integration tasks. The advantages of the three-layer design are described in more detail in Section 3.4.

In order to support type system evolution, most of the typechecking presented here is designed to be invariant with respect to the *covariant transformations* described in Section 5.6.1. One of the consequences of this feature is the localization of function code typing. A change in a function code does not affect the typing or validity of the program code outside of the association that uses the modified function. This is in contrast with ML-style type inference systems where a local change in a function code can potentially affect the typing of the code outside the function. The difference lies in the presence of explicit type annotations of behavior definitions. While type inferencing is still possible in the presented system, its scope is restricted to a single association.

The application of the principles and constructs described in this chapter is given in Section 3.5, where an example of a basic type system designed for the TIGUKAT object model [Pet94] is discussed.

3.1 Types and behaviors

The notion of a *type* plays a central role in any type system. This section is devoted to types, their flavors, and the type specification issues. Special attention is given to the issues related to subtyping and type safety.

Types are introduced in Section 3.1.1. Then, in Section 3.1.2, the notion of *subtyping* is discussed. Section 3.1.3 offers a detailed discussion of behavior specifications, redefinition, and inheritance. Sections 3.1.4 through 3.1.9 discuss various type flavors present in the type system. Additional flexibility of constrained type specifications is discussed in Section 3.1.10. Interaction between subtyping and inheritance is considered in Section 3.1.11. Section 3.1.12 concludes the discussion of this layer of the type system.

3.1.1 The notion of a type

In the type system presented here, *type* denotes a concept. The concept might originate from the real world (such as "person" or "student"), mathematics ("integer number" or "set"), computer science ("output stream" or "character"), or any other domain. For instance, a statement "5 is an integer number" can be rephrased as "`5` has type `T_Integer`"[1]. In the same way, it is possible to say "`john` has type `T_Person`" meaning that "John is a person".

Each object in a program has a type. The type not only characterizes the conceptual properties of an object, it also defines its programmatic interface. The presented type system is based on a *uniform behavioral object model* [Pet94]. In this model, everything is an object, and the only way to manipulate an object is via *behavior applications* (message sends, function applications). Therefore the statement that the type defines a programmatic interface for its objects is equivalent to the statement that the type defines a set of behaviors applicable to a particular object. The type can also define implementations (methods) for one or more of its behaviors.

Consider, for example, type `T_Person`. Objects of this type have an age, which is represented as a natural number, a name (represented as a string), and possibly a spouse (another person). The definition of a person type will have the following format:

```
type T_Person {
    age() : T_Natural;
    name() : T_String;
    spouse() : T_Person;
};
```

[1] In the rest of the dissertation, types will be prefixed by `T_`, classes by `C_`, and implementation types by `IT_`. Also, a `typewriter font` will be used to denote objects that occur in the program, while normal font will be used to denote the real-world objects and concepts they are designed to model. This convention will be followed through the rest of the dissertation, unless otherwise specified.

Given this definition, the following behavior application is legal provided the type of **john** is **T_Person**: **john.age()**. It will return an object of type **T_Natural**.

While simple, this notion of a type has certain non-trivial implications. Some of them manifest themselves when *subtyping* is considered; they are discussed later. An implication that is immediate is the absence of object structure definition in a type. Unlike many other type systems, the type system discussed here totally separates object interface from the object structure. An object of type **T_Person** might be implemented as a list of three slots, holding age, name, and a spouse; it might also be implemented as a list of five slots holding date of birth, first, second, and middle names, and a spouse. Both objects can legally and simultaneously belong to the type **T_Person** as long as they provide the interface defined in it. The advantages of such separation, as well as a more detailed description of the mechanisms that achieve it, will be given in Section 3.2.

Another implication of the notion of a type adopted here is the fact that a type is more than just an interface. In other words, two types can have identical interfaces, and still be considered unrelated to each other. Such an arrangement is termed as *name equivalence*, as opposed to *structural equivalence*. The structural equivalence principle states that two types are equivalent and indistinguishable from each other if they have identical interfaces. In order to illustrate the difference between the two approaches, let us consider the following example.

```
type T_Person {
    age() : T_Natural;
    name() : T_String;
};
type T_Wine {
    age() : T_Natural;
    name() : T_String;
};
```

Here, types **T_Person** and **T_Wine** have identical interfaces. Under the rules of name equivalence, these two types are distinct and unrelated; an attempt to bind a variable declared to be of type **T_Person** to an object of type **T_Wine** will be considered a compile-time error. On the other hand, if structural equivalence rules were in force, such an assignment would be considered legal, since in this case both **T_Person** and **T_Wine** would be regarded as two different aliases for the same entity (the interface). The choice between name equivalence and structural equivalence is a design decision. Structural equivalence is more permissive as it allows code sharing between semantically unrelated types. On the other hand, sometimes structural equivalence is "too permissive", as in the above example. Most of the "real" languages today adopt name equivalence as the safer and more transparent alternative (Table 2.3). Name equivalence and structural equivalence are two opposite ends of a spectrum of possible design decisions; some intermediate approaches are also possible. An example of such an intermediate approach is the notion of semantical, or behavioral, equivalence [NP91, LW94, LW95]. In this approach, equivalence is defined in terms of "observable behavior". While quite intuitive and appealing, this approach suffers from a major drawback: the notion of behavioral equivalence is undecidable in general.

3.1.2 Subtyping

The power of a type system comes in part from its ability to adequately model an application domain. In the real world, concepts are not independent of each other. For example, each student is also a person; an integer number can also be considered as a special case of a real number. In order to model this relationship, the type system provides a notion of *subtyping*. If a type **T_Person** models the concept of a person, a type **T_Student** models the concept of a student, and each student is also a person, then the type **T_Student** is called a *subtype* of the type **T_Person** (denoted **T_Student** $\preceq$ **T_Person** or **type T_Student subtype of T_Person**).

What does it mean for a type to be a subtype of another one at a programming language level? Since every object that belongs to the first type also belongs to the second (*subsumption property*; a student is a person), it is natural to assume that an object of a subtype can be legally used everywhere an object of its supertype can. This is termed *the substitutability property* and considered to be one of the most powerful mechanisms for code reuse in object-oriented programming.

In the following example, a type **T_Student** is defined as a subtype of the type **T_Person**. An additional behavior **studentId** is defined for this type.

```
type T_Person {
    age() : T_Natural;
    name() : T_String;
    spouse() : T_Person;
};
type T_Student subtype of T_Person {
    studentId() : T_Natural;
};
```

In the presence of these declarations, the behavior applications **aStudent.age** and **aStudent.studentId** are valid[2], while **aPerson.studentId** is not (every student is also a person, but not every person is a student).

The term *subtyping* is sometimes used to denote *inheritance*. Yet there is a considerable difference between these two notions. *Inheritance* is the ability to reuse properties of a type (interface, code, structure, or any combination of the above) in another type. Inheritance does not imply subtyping, but subtyping implies interface inheritance. An inheriting type may or may not be a subtype of the type it inherits from. Thus, inheritance is more general than subtyping. On the other hand, subtyping guarantees substitutability; inheritance does not. This is discussed further in Section 3.1.11.

The subtyping relationship between types defines a partial order on the domain of types. If there are two types T_A and T_B such that $T\_A \preceq T\_B \wedge T\_B \preceq T\_A$, then $T\_A = T\_B$. Since subtyping is user-defined, the system has to verify that this condition is satisfied. The presence of parametricity makes such checking a non-trivial task, as discussed further in Section 3.1.7.

A type can subtype several types (*multiple subtyping*). This feature improves the expressibility of the type system and allows straightforward definitions of complicated type hierarchies. Consider, for example, the following type hierarchy:

```
type T_Person {
    age() : T_Natural;
    name() : T_String;
    spouse() : T_Person;
};
type T_Student subtype of T_Person {
    studentId() : T_Natural;
};
type T_Teacher subtype of T_Person {
    coursesTaught : T_Set(T_Course);
};
type T_TeachingAssistant subtype of T_Student, T_Teacher;
```

Here, a teaching assistant is both a student and a teacher. A teaching assistant is also a person, so age, name, and spouse apply. Being also a student, there is a student id; being a teacher, there is a set of courses taught.

While the situation described here is not uncommon in practice, many type systems do not allow multiple subtyping (Table 2.3). The major reason for this restriction is the fact that multiple subtyping incurs significant complications for systems and languages where subtyping is coupled with inheritance of code and structure. Multiple subtyping also makes type system theory and dispatch significantly more complicated.

[2]It is assumed that **aStudent** evaluates to an object of type **T_Student**, while **aPerson** evaluates to an object of type **T_Person**.

The type system presented in this dissertation has multiple subtyping, as it increases the modeling power of the system. Its theoretical treatment will be discussed in Chapter 5.

When a type system is used to design an application, it is often necessary to add new supertypes, subtypes, or behaviors to existing types. It is often desirable, and sometimes required, to do these modifications *without changing the original type specifications.* Most of today's object-oriented languages allow addition of subtypes without touching the existing code. However, addition of new behaviors and/or supertypes[3] can not be done without changing the original specifications.

The presented type system is designed for database programming languages where the problem of evolution is even more important then in traditional languages. The ability to reuse legacy code while providing new types and behaviors is therefore crucial for the type system.

The method that is used in the presented type system is based on *specification combination*: there can be more then one specification of a single type, and they are all glued together automatically during compilation. The same is true of all other specifications used in the presented type system.

Consider the situation when a type **T_Employee** has been defined in a university payroll. Later on, new types **T_Student** and **T_TeachingAssistant** have to be introduced into the system because it is being expanded to handle student payments as well. The integrated application will have to produce sets where both employees and students are present; moreover, some behaviors previously defined for employees (such as **salary**) have to handle students as well. The following is the existing specification:

```
type T_Employee {
    salary() : T_Amount implementation ...
    ...
};
```

Using the specification combination method, the following code has to be added:

```
type T_Person {
    name() : T_String;
    salary() : T_Amount;
    printSalary() implementation { name.print; salary.print; };
};
type T_Employee subtype of T_Person;
type T_Student subtype of T_Person {
    salary() : T_Amount implementation ...
    ...
};
type T_TeachingAssistant subtype of T_Student, T_Employee;
```

The addition of this code to the code above will make **T_Person** a common supertype of the old type **T_Employee** and the new type **T_Student** (Figure 3.2) making it possible to operate on sets of persons; for example, the following code would print salary information for all people in the database, provided that **allStudents** evaluates to the set of all students and **allEmployees** evaluates to the set of all employees:

```
T_Set(T_Person) allPeople := allStudents.union(allEmployees);

allPeople.foreach(printSalary);
```

If this mechanism was not available, the common supertype **T_Person** could not be created. Then, the code to print all names and salaries would not be able to use **union** to automatically filter out the duplicated teaching assistants, and extra code would have to be written to support this operation.

[3]The importance of the ability to create new supertypes of existing types has been discussed in [Höl93b] in the context of interoperability.

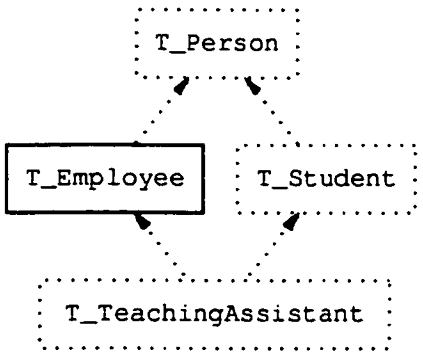

Figure 3.2: Person type hierarchy

3.1.3 Interface specification and product types

The specification of the programmatic interface is the main component of a type specification. In the previous sections, some examples of interface specification have already been considered. In the type system presented, the interface consists of a set of *behavior specifications*. Each behavior specification lists behavior argument types and specifies the return type. For example, the following specifies the type **T_Real** and its interface that consists of six behavior specifications:

```
type T_Real {
    add(T_Real) : T_Real;
    subtract(T_Real) : T_Real;
    multiply(T_Real) : T_Real;
    divide(T_Real) : T_Real;
    negate() : T_Real;
};
```

The above specification defines the standard arithmetic operations.

Consider the following example where the type **T_ChequingAccount** is defined. A customer can deposit or withdraw cash or cheques to/from a chequing account and receive a transaction record.

```
type T_ChequingAccount {
    deposit(T_Cheque) : T_TransactionRecord;
    deposit(T_Cash) : T_TransactionRecord;
    withdraw(T_Cheque) : T_TransactionRecord;
    withdraw(T_Cash) : T_TransactionRecord;
};
```

Here, the behavior **deposit** has two different interface specifications. Multiple interface specifications are allowed as long as they differ in their argument types. A distinction in return types only is disallowed; for example, the following specification is invalid:

```
type T_Account {
    deposit(T_Cheque) : T_TransactionRecord;
    deposit(T_Cheque);
};
```

In the second case the return type is implicitly assumed to be a special predefined type **T_Unit**, which is analogous to type **void** in C++ [Str91] and Java [AG96].

In order to see why return type differences are insufficient, consider the following example:

```
type T_A {
    beh() : T_C;
    beh() : T_D;
};
type T_C subtype of T_B;
type T_D subtype of T_B;
...
T_A a;
T_B b;
...
b := a.beh();
```

Since at run-time the object b may have dynamic type T_C or T_D, it is unclear which behavior specification to use. Both are legal, and none is more specific then the other. In order to avoid this kind of ambiguity, it is required that behavior specifications be different in their argument types.

When a subtype is defined, the interface specification inherited from its parent can be modified. These modifications should follow *the return type covariance rule* in order for the newly defined type to be a subtype of its parent. The rule states[4] that the return type in a behavior specification for a subtype should be a subtype of the return type in the specification of the same behavior in its parent type. For example, the specification[5]

```
type T_Real;
type T_Integer subtype of T_Real {
    succ() : T_Integer;
};
type T_SmallInteger subtype of T_Integer;

type T_Person {
    age() : T_Integer;
};
type T_Child subtype of T_Person {
    age() : T_SmallInteger;
};
```

is allowed, while the specification

```
type T_Person {
    age() : T_Integer;
};
type T_Child subtype of T_Person {
    age() : T_Real;
};
```

is not. In order to see why this last specification is considered illegal, consider the following code fragment:

```
T_Person p;
T_Integer i;
...
p := aChild;       // 1 (the type of aChild is T_Child)
...
i := p.age;       // 2
...
i := i.succ;      // 3
```

This code is type-correct: the first assignment is type-correct, since the static type of p is T_Person and the static type of aChild is T_Child which is declared to be a subtype of T_Person. The second

[4]A more precise formulation of this rule will be considered later in this section

[5]Note that this specification is equivalent to an implementation of the test (*PERSON*) from the test suite given in Section 2.6.

assignment is type-correct since the return type of **age** defined on the type **T_Person** is **T_Integer**. The third assignment is also type-correct as the return type of **succ** is **T_Integer**. On the other hand, at run-time the second assignment will put a real number (of type **T_Real**) into the variable **i** since the behavior specification of **age** on the type **T_Child** has return type **T_Real**. The call to **succ** in the third assignment will then fail with a run-time error, since the behavior **succ** is not defined on **T_Real**.

Note that no restrictions are placed on the *argument* types. In other words, the following definition is legal:

```
type T_Real {
    add(T_Real) : T_Real;
};
type T_Integer subtype of T_Real {
    add(T_Integer) : T_Integer;
    succ() : T_Integer;
};
```

While this definition seems to be natural, its legality requires some explanation.

A significant number of object-oriented languages today use *single dispatch*, in which the method to execute is chosen according to the type of the receiver (Table 2.3). In such languages, the above specification (with covariant arguments) would not be type-safe. In order to see why, consider the following code fragment:

```
type T_Point {
    getX() : T_Real;
    getY() : T_Real;
    compare(T_Point x) : T_Boolean implementation {
        return     x.getX == getX
               and x.getY == getY ; };
};
type T_ColorPoint subtype of T_Point {
    getColor() : T_Color
    compare(T_ColorPoint x) : T_Boolean implementation {
        return     x.getX == getX
               and x.getY == getY
               and x.getColor == getColor ; };
};
...
T_Point p1, p2;
...
p1 := aPoint;
p2 := aColorPoint;
...
p2.compare(p1);
```

If this type specification was considered to be legal, then the code following it would be type-correct; assignments are type-correct since **T_ColorPoint** is declared to be a subtype of **T_Point**, and the application of **compare** is type-correct since it compares two points. The methods that implement comparison are also type-correct since they only rely on behaviors that are defined on their argument types. However, at run-time the application of **compare** will be dispatched to the method defined for color points (according to the type of the receiver, which is **T_ColorPoint**) and it will fail at run-time, since its argument is of type **T_Point** which does not define the behavior **getColor**.

Therefore, in all type-safe single-dispatched languages covariant specification of argument types is disallowed. However, such specification is natural and desirable. In order to use covariant argument type specification safely, *multiple dispatch* is required.

In languages with multiple dispatch, the method to invoke is determined according to the run-time types of *all* arguments, not just the receiver. In such languages, the above example with points and color points would not fail at run-time, because the behavior application in question would be dispatched to the method defined for points.

The main idea behind multiple dispatch is the uniform treatment of all arguments. While in single-dispatching languages the receiver has a special status, multiply-dispatching languages treat all arguments as a single unified receiver. This concept can be neatly described by using the notion of a *product type*. A product type is a tuple of the form $\langle T_1, \dots, T_n \rangle$, where T_i are simple (non-product) types. Subtyping between product types is induced by the subtyping between simple types and is defined as

$$\langle T_1, \dots, T_n \rangle \preceq \langle Q_1, \dots, Q_m \rangle \overset{\text{def}}{\Longleftrightarrow} (m = n \wedge T_i \preceq Q_i \text{ for all } i = 1, \dots, n) \qquad (3.1)$$

Now, every behavior can be considered as having a single argument — the tuple of all arguments defined for it, the first being the original receiver. Since every behavior is always applied to a product, angle brackets are unnecessary and can usually be omitted.

It is possible to translate the standard object-oriented notation for behavior specification and application into the product-type notation. The product-type notation is mainly used for theoretical purposes, but it is also useful for the description of multiple dispatch. The following code fragment is a product-type translation of the point example above[6]:

```
type T_Point;

getX(T_Point) : T_Real;
getY(T_Point) : T_Real;
compare(T_Point r, T_Point x) : T_Boolean implementation {
        return     getX(x) == getX(r)
               and getY(x) == getY(r) ; };

type T_ColorPoint subtype of T_Point;

getColor(T_ColorPoint) : T_Color;
compare(T_ColorPoint r, T_ColorPoint x) : T_Boolean implementation {
        return     getX(x) == getX(r)
               and getY(x) == getY(r)
               and getColor(x) == getColor(r) ; };
...
T_Point p1, p2;
...
p1 := aPoint;
p2 := aColorPoint;
...
compare(p2, p1);
```

Product-type notation resembles a procedural notation; however, this is only a superficial similarity. Here, the behavior **compare** has methods on two product types: ⟨**T_Point**, **T_Point**⟩ and ⟨**T_ColorPoint**, **T_ColorPoint**⟩. The run-time argument type for the behavior application **compare(p2, p1)** is ⟨**T_ColorPoint**, **T_Point**⟩, which is a subtype of ⟨**T_Point**, **T_Point**⟩, but not a subtype of ⟨**T_ColorPoint**, **T_ColorPoint**⟩, according to Equation 3.1. Therefore, the method to execute is the one defined for points.

Another illustrative example of product type dispatch is the dispatch of the behavior application **p2.compare(p2)** which will be translated to product-type notation as **compare(p2,p2)**. Here, the run-time type of the argument is ⟨**T_ColorPoint**, **T_ColorPoint**⟩, which is a subtype of both ⟨**T_ColorPoint**, **T_ColorPoint**⟩ (1) and ⟨**T_Point**, **T_Point**⟩ (2). However, since the product type

[6]This is an implementation of the test (*POINT*) from the test suite given in Section 2.6.

(1) is a subtype of the product type (2), the method associated with the product type (1) is chosen as the more specific one. Thus, the behavior application **p2.compare(p2)** will execute the code written for color point comparison, as expected.

Having reviewed product type notation and dispatch, we will now return to the specification of real and integer numbers. Rewritten in a product-type notation, the example specification is:

```
type T_Real;

add(T_Real, T_Real) : T_Real;
subtract(T_Real, T_Real) : T_Real;
multiply(T_Real, T_Real) : T_Real;
divide(T_Real, T_Real) : T_Real;
negate(T_Real) : T_Real;

type T_Integer subtype of T_Real;

add(T_Integer, T_Integer) : T_Integer;
subtract(T_Integer, T_Integer) : T_Integer;
multiply(T_Integer, T_Integer) : T_Integer;
negate(T_Integer) : T_Integer;
succ(T_Integer) : T_Integer;
```

Here, there is no code. However, in a product type world, not only the *code* to execute, but also the *specification* to use for typechecking is chosen according to the product type of the receiver. Consider the following code:

```
T_Integer i1, i2;
...
i1.add(i2).succ;
```

In the product notation, the behavior application above will be **succ(add(i1, i2))**. Since the static type of the tuple **<i1, i2>** is ⟨**T_Integer**, **T_Integer**⟩, both of the specifications **add(T_Real, T_Real) : T_Real** and **add(T_Integer, T_Integer) : T_Integer** are applicable to it. However, the second one is more specific than the first and is therefore chosen for typechecking. The result type of the second specification is **T_Integer**, and since the behavior **succ** is defined on this type, the whole expression is considered to be type-correct.

On the other hand, the following code will produce a type-checking error:

```
T_Integer i;
T_Real r;
...
i.add(r).succ;
```

In this case, the call in question will translate to **succ(add(i, r))**, and the static type of **<i, r>** is ⟨**T_Integer**, **T_Real**⟩. Therefore, of the two specifications for the behavior **add**, only the first one will do: the second requires that the receiver type be a subtype of ⟨**T_Integer**, **T_Integer**⟩, and the type of **<i, r>** is not. The first specification has a return type **T_Real**, but the behavior **succ** is not defined on **T_Real**, and the typechecking reports an error. This is the expected result as adding an integer to a real yields a real number, and the successor function is not defined on reals.

The above specification of types **T_Integer** and **T_Real** can be significantly simplified by the use of **selftype**[7], which is a reference to the defining type that changes covariantly along the type hierarchy. The following is a (non-product) specification that uses **selftype**:

```
type T_Real {
    add(selftype) : selftype;
```

[7] A more formal treatment of **selftype** will be given in Section 3.1.10

```
        subtract(selftype) : selftype;
        multiply(selftype) : selftype;
        divide(selftype) : T_Real;
        negate() : selftype;
    };

    type T_Integer subtype of T_Real {
        succ() : T_Integer;
    };
```

Note that the result type of the behavior **divide** is specified as **T_Real**, not as **selftype**. If it was specified a **selftype**, the inherited definition in **T_Integer** would be

```
    type T_Integer subtype of T_Real {
        divide(selftype) : selftype;
    };
```

which means that an integer divided by an integer is always an integer, which is not the case. Therefore, the result type of **divide** has not been changed to **selftype**.

Usage of **selftype** in argument types is known to be statically type-unsafe in languages with single dispatch [Cas95a, BG96]. However, it is so convenient that some language designers chose to drop static type safety in favor of it (e.g., Eiffel [Mey88]), while others chose to drop substitutability (e.g., LOOM [BFP96]). However, in a multiply-dispatching language covariant specification is type safe, so it is possible to have it all: use of **selftype**, type safety, and substitutability [Cas95a, BG96].

With the introduction of product type notation, it is now possible to give a precise formulation for the return type covariance rule discussed at the beginning of this section. The rule is as follows: for any two specifications of the same behavior, if the product receiver type of the first behavior specification is a subtype of the product receiver type of the second specification, then the return type of the first specification should be a subtype of the return type of the second. Thus, the following is allowed:

```
    type T_Cash;
    type T_SmallAmountOfCash subtype of T_Cash;
    type T_TransactionRecord;
    type T_SimplifiedTransactionRecord subtype of T_TransactionRecord;
    type T_Account {
        deposit(T_Cash) :  T_TransactionRecord;
        deposit(T_SmallAmountOfCash) : T_SimplifiedTransactionRecord;
    };
```

This would not be allowed if **T_SimplifiedTransactionRecord** was not declared to be a subtype of **T_TransactionRecord**, since in product notation the two behavior specifications would read

```
    deposit(T_Account, T_Cash) :  T_TransactionRecord;
    deposit(T_Account, T_SmallAmountOfCash) : T_SimplifiedTransactionRecord;
```

Since ⟨**T_Account**, **T_SmallAmountOfCash**⟩ $\preceq$ ⟨**T_Account**, **T_Cash**⟩, it should also be true that **T_SimplifiedTransactionRecord** $\preceq$ **T_TransactionRecord** (otherwise the rule would be violated).

In this section, the interface specifications were discussed. The problem of behavior redefinitions was considered and its solution was outlined. The notions of *multiple dispatch* and *product type notation* were introduced and used to illustrate the mechanisms behind the behavior redefinition semantics adopted in the presented type system.

3.1.4 Behavior types

Before the introduction of the object-oriented programming paradigm, *data types* were considered as object structure definitions. This point of view is present in procedural programming languages such as Pascal [JW75], as well as in database programming where the set of data structure definitions for a database is called its *schema*. As the object-oriented programming paradigm gained popularity, the focus of data (object) specifications has been slowly shifting from object structure to object functionality. In the object-oriented programming model, the functionality is the major characteristic, while the object structure plays a secondary role.

An object structure specification describes the logical interpretation of the physical storage the object occupies. On the other hand, specification of object functionality focuses on the things one can do with a particular object, i.e. the set of actions (behaviors, messages, functions) that can be performed (applied) on a particular object.

A unit of structure specification is therefore a *field definition*. A field definition describes both the amount of physical memory allocated for a field and its logical treatment. At the same time, a unit of functionality specification is an *action* (behavior, message). Here, the action *declaration* (behavior specification) can be viewed as being parallel to the logical field type (abstract interpretation), while the action *definition* (method, function code) can be considered parallel to the physical layout of a particular field (concrete interpretation). Thus, the relationship between an action declaration and an action definition is parallel to the relationship between field type and its physical layout.

It is therefore natural to consider the behavior specification as a type of the behavior for which such a specification is given. In other words, when a behavior **b** is specified as being applicable to a certain type **T_t**, such specification contains information about both the type **T_t** and the behavior **b**. Note that the parallel between actions and fields goes even further: the simplest possible concrete interpretation of fields (binary bits) is capable of representing any information via binary numbers, while the simplest possible concrete interpretation of actions (λ-terms) is capable of representing any computation via λ-calculus [Tur37].

When behavior specifications are considered as types, behaviors themselves become objects. Such an arrangement leads to a truly uniform model, in which all entities, including actions, are objects. This allows one to extend the notions of subtyping and substitutability to behavior types and behaviors. The standard notation for behavior types is **T_a→T_r**, where **T_a** is an argument type, while **T_r** is the result type of a behavior. For example, the type **T_Real→T_Real** is the type of functions that produce a real result when applied to a real number. In the example syntax adopted here, the same behavior type can also be written as **(T_Real):T_Real**, which is parallel to the syntax of behavior specifications.

Consider the following specification:

```
type T_Real {
    negate(): T_Real;
}
...
const T_Real r;
```

which is equivalent to

```
type T_Real;
negate(T_Real): T_Real;
...
const T_Real r;
```

in product-type notation. This can be considered as an abbreviation for

```
type T_Real;
const (T_Real):T_Real negate;
...
const T_Real r;
```

that defines two typed constants: **negate** of type **(T_Real):T_Real** and **r** of type **T_Real**. This last specification is in turn equivalent to

```
type T_Real;
const (T_Real -> T_Real) negate;
...
const T_Real r;
```

Behavior types are not atomic: they are composed from other types using the behavior type constructor $\rightarrow$ (or, equivalently, `()`:), much the same way as product types are composed of other types using the type constructor $\langle\rangle$. Therefore, it is relevant to define the subtyping rules for the behavior types based on the subtyping relationship between their components. The general subtyping rule for behavior types is as follows:

$$(A_1 \rightarrow R_1 \preceq A_2 \rightarrow R_2) \overset{\text{def}}{\Longleftrightarrow} (A_2 \preceq A_1 \wedge R_1 \preceq R_2) \tag{3.2}$$

Note that the relationship between argument types is reversed[8]. In order to see why this is the case consider the following example:

```
type T_Real {
    truncate() : T_Integer;
};
type T_Integer subtype of T_Real {
    succ() : T_Integer;
};
...
(T_Real):T_Integer rifunc;
(T_Integer):T_Integer iifunc;
...
iifunc := truncate;
5.iifunc;
...
rifunc := succ;              // Compile-time type error
(5.0).rifunc;
```

Here, two variables **rifunc** and **iifunc** are defined. The first one is declared to be of type **T_Real**$\rightarrow$**T_Integer**, while the second one is defined to be of type **T_Integer**$\rightarrow$**T_Integer**. Constants (behaviors) **truncate** and **succ** are defined to be of types **T_Real**$\rightarrow$**T_Integer** and **T_Integer**$\rightarrow$**T_Integer**, respectively. The first assignment in the above code is statically type-correct since

type of **truncate** = **T_Real**$\rightarrow$**T_Integer** $\preceq$ **T_Integer**$\rightarrow$**T_Integer** = type of **iifunc**

according to the Equation 3.2. The subsequent application of the behavior stored in the variable **iifunc** to the integer constant **5** is also statically type-correct, since the argument type of **iifunc** is declared to be **T_Integer**. When this behavior application is executed, **truncate** is applied to an integer. This is type-correct because **truncate** is declared to operate on any real number, and an integer is a real.

On the other hand, the assignment of **succ** to the variable **rifunc** is *not* statically type-correct, since it is not true that

type of **succ** = **T_Integer**$\rightarrow$**T_Integer** $\not\preceq$ **T_Real**$\rightarrow$**T_Integer** = type of **rifunc**

If this assignment was allowed, the subsequent application of **rifunc** to the real number **5.0** would fail at run-time, as the behavior **succ** is not designed to work on real numbers.

So far a single behavior specification was used in the examples to specify a type of a behavior. However, a behavior can have several specifications (Section 3.1.3). In this case, the type of a behavior is defined as *the greatest lower bound* (glb) of the types given by these specifications.

Consider the following specification:

[8]This fact is usually formulated as follows: The type constructor $\rightarrow$ is contravariant in its first position.

```
type T_Real {
    add(T_Real) : T_Real;
};

type T_Integer subtype of T_Real {
    add(T_Integer) : T_Integer;
};
```

or, in product type notation with behavior types,

```
type T_Real;
const (T_Real, T_Real)->T_Real add;

type T_Integer subtype of T_Real;
const (T_Integer, T_Integer)->T_Integer add;
```

The two specifications of the behavior constant **add** are combined, and the resulting specification is

```
type T_Real;
type T_Integer subtype of T_Real;

const glb((T_Real, T_Real)->T_Real, (T_Integer, T_Integer)->T_Integer) add;
```

Thus, the resulting type of **add** is

$$\mathsf{glb}(\langle \mathtt{T\_Real}, \mathtt{T\_Real}\rangle \rightarrow \mathtt{T\_Real}, \langle \mathtt{T\_Integer}, \mathtt{T\_Integer}\rangle \rightarrow \mathtt{T\_Integer})$$

This expression can be understood as follows: the behavior **add** produces a real result when applied to a pair of real numbers, and an integer result when applied to a pair of integer numbers. A behavior application `5.add(6)` produces an integer result, while behavior applications `5.add(6.0)`, `(5.0).add(6)`, and `(5.0).add(6.0)` produce real results. This is so because only the product-type of the argument of the first behavior application is a subtype of ⟨**T_Integer**,**T_Integer**⟩, while all four of the product-types of the arguments of these applications are subtypes of ⟨**T_Real**,**T_Real**⟩. Therefore, the results of all four applications are known to be subtypes of **T_Real**, while the result of the first behavior application is additionally known to be a subtype of **T_Integer**.

Consider the following piece of code (provided that the above specification of the behavior **add** is in effect):

```
(T_Real, T_Real):T_Real rrfunc;
(T_Integer, T_Integer):T_Integer iifunc;
...
rrfunc := add;
(5.0).rrfunc(6.0);
...
iifunc := add;
5.iifunc(6);
```

When the assignment of the behavior constant **add** to the variables **rrfunc** and **iifunc** is type-checked, the *combined* type of the behavior **add** is taken into account. Therefore, both assignments type-check:

Let $\mathtt{T\_RR} \equiv \langle \mathtt{T\_Real}, \mathtt{T\_Real}\rangle \rightarrow \mathtt{T\_Real}$ and $\mathtt{T\_II} \equiv \langle \mathtt{T\_Integer}, \mathtt{T\_Integer}\rangle \rightarrow \mathtt{T\_Integer}$. Then

$$\begin{aligned} \text{type of } \mathbf{add} &= \mathsf{glb}(\mathtt{T\_RR}, \mathtt{T\_II}) \preceq \mathtt{T\_RR} = \text{ type of } \mathtt{rrfunc} \\ \text{type of } \mathbf{add} &= \mathsf{glb}(\mathtt{T\_RR}, \mathtt{T\_II}) \preceq \mathtt{T\_II} = \text{ type of } \mathtt{iifunc} \end{aligned}$$

Note that in the absence of greatest lower bounds the above code would not type-check: if the type **T_RR** was assigned to the behavior **add**, then the assignment `iifunc := add` would not typecheck;

on the other hand, using the type T_II for the type of add would not allow us to type-check the assignment rrfunc := add. Only the combination of these two types in the form of the greatest lower bound enables successful typechecking of both assignments.

In this section, the notion of a *behavior type* has been introduced. Properties of behavior types and their usage were discussed, and the principle of combining behavior types from different specifications using the greatest lower bound was considered. All these notions will be given a formal treatment in Chapter 5.

3.1.5 Mutable types

The ability to define any computation in the applicative (functional, stateless) paradigm has led to the development of a number of programming languages. These languages share several important properties: strict and rigorously defined semantics, simplicity of the main programming paradigm, and well-established theoretical mechanisms for reasoning about programs. On the other hand, the biggest disadvantage of these languages from a practical perspective is exactly the same feature that makes them so attractive from the theoretical viewpoint: the inability to deal with *state*.

While there are certain applications that can be programmed without using the concept of state, most real-world applications rely on the ability of a program to store and retrieve data. This is especially true of database applications, as database data have a life span beyond the execution time of a program. Therefore if a program is unable to deal with state change, the data in a database can never be updated.

The ability of a type system to deal with mutable data is essential for database programming. However, many languages with well-developed and theoretically sound type systems either do not consider mutable data at all or severely restrict their use (Section 2.7).

The ML language [MTH90] is a representative example of the languages in the last category. It is based on the functional programming paradigm, but provides some imperative features in the form of its *reference type constructor* ref. However, due to the problems related to typing of imperative features in a functional environment, the use of the *reference cells* in ML is much more restricted than the use of all other type constructors. An excellent review of this problem is given in [App92].

In imperative object-oriented programming languages, the concept of state is usually captured by two separate but related notions: the notion of an *attribute* and the notion of a *variable*. An *attribute* is a slot inside an object that is capable of holding an object (or an object reference) of a particular type. An object can be put into an attribute (*assigned to* it) or extracted from it. A *variable* is similar to an attribute in its ability to hold an object. On the other hand, while an attribute is a part of an object, a variable is not.

Here, only the issues related to typing of behaviors that are used to access or set attributes will be discussed. The discussion of the other aspects of state (such as typing of local variables, definition of object structure using a set of attributes and its relationship to the behavioral interface, et cetera) appears in Section 3.2.

In the proposed type system, setting an attribute (or a variable) is an ordinary behavior application. The assignment syntax **aPerson.age := 45** is considered as an abbreviation for **aPerson.set_age(45)** which is, in turn, an abbreviation for **set_age(aPerson, 45)**. The attribute specification in this system belongs to a *class*, not a *type*; the type only specifies the interface for getting and setting the attributes. This corresponds to the absence of public instance variables. The advantages of this approach will be discussed later in Section 3.2. Here a familiar notation for specifying an attribute interface as a part of a type will be introduced and the typing implications of this mechanism will be discussed.

The following specification

```
type T_Person {
    age() :=: T_Natural;
    name() :=: T_String;
};
```

specifies a type `T_Person` that has two assignable "attributes": a name and an age. Note here the sign `:=:` that is used to separate the behavior arguments from its result type. It is a combination of the sign `:` used to specify the return type for immutable behaviors and the sign `:=` used for assignment. The specification `age() := T_Natural` is also possible; it means that a person's age can not be read, but it can be assigned to.

The above specification makes the following behavior applications legal:

```
T_Person p;

p.age := p.age.add(5);
p.name := "John";
```

The type specification for `T_Person` given above is equivalent to

```
type T_Person;

age(T_Person) : T_Natural;
set_age(T_Person, T_Natural);
name(T_Person) : T_String;
set_name(T_Person, T_String);
```

and the behavior applications are translated into

```
T_Person p;

set_age(p, add(age(p), 5));
set_name(p, "John");
```

Thus, the standard product-type typing rules described in Section 3.1.3 apply to assignment as well.

A more complicated example of user-defined assignment is the definition of the behavior `at` that is capable of getting the n-th element of a list. It is also possible to assign to an element, as in the following code[9]:

```
type T_MutableList {
    at(T_Natural) :=: T_Object;
};
...
T_MutableList mlist;
...
mlist.at(4).print;
mlist.at(4) := object;
```

The translation of this code fragment is

```
type T_MutableList;
at(T_MutableList, T_Natural) : T_Object;
set_at(T_MutableList, T_Natural, T_Object);
...
T_MutableList mlist;
...
print(at(mlist, 4));
set_at(mlist, 4, object);
```

The first behavior application prints the fourth element of an array, while the second behavior application sets it to `object`. Thus, the treatment of assignment proposed here allows the programmer

[9]It is more precise to define `T_MutableList` as a *parametric type* parameterized by the type of its elements. Such a specification will be considered in Section 3.1.6.

to specify custom versions of the assignment operation, including the possibility of using additional arguments.

The standard subtyping rule for updatable attributes requires their typing to be novariant. The following examples illustrate the reasons behind this requirement, while the subsequent discussion demonstrates how the attribute novariance requirements are relaxed in the proposed type system.

Consider the following specification:

```
type T_SmallNatural subtype of T_Natural;
type T_Person {
    age() :=: T_Natural;
};
type T_Child subtype of T_Person {
    age() :=: T_SmallNatural;
};
```

Two types are specified, **T_Person** and **T_Child**, with an updatable attribute **age**. A person's age is a natural number, while a child's age is a small natural number (a subtype of natural). If the attribute **age** was immutable, such an arrangement would be perfectly legal. However, the mutable nature of this attribute causes a problem that manifests itself in the typing of the following code:

```
T_Person aPerson;
T_Child aChild;
T_Natural n;

aPerson := aChild;
aPerson.age := n;
```

The above code is type-correct. However, after its execution the **age** attribute of **aChild** will contain a natural number rather than a small natural one, which contradicts the specification of the attribute **age**. Situations like the one presented in this example are the primary reason for the novariance requirement for mutable attributes in current object-oriented type systems.

On the other hand, there are situations in which the novariance requirement is too strict. Consider the following specification:

```
type T_CharacterList {
    substring(T_Natural start, T_Natural end) :=: T_CharacterList;
};
type T_String subtype of T_CharacterList {
    substring(T_Natural start, T_Natural end) :=: T_String;
    translateFromFrench() : T_String;
};
```

It allows for the following code[10] to be successfully typechecked:

```
T_String string;
T_CharacterList list;

string := "pteit mluot";
string.substring(8,9) := "ul"; // The resulting string
                               // will contain "pteit mulot"
list := ('e', 't');
string.substring(2,3) := list; // The resulting string
                               // will contain "petit mulot"
string.substring(1,5).translateFromFrench; // The result will be the
                               // translation of "petit" , i.e. "small"
```

[10]It is assumed that numbering of arrays and strings starts from 1, not from 0.

If the novariance requirement was in force, the redefinition of **substring** in the type **T_String** would be disallowed. Then, the last behavior application that performs the translation would not type-check, as the result type of **substring** would be considered to be **T_CharacterList** which does not have behavior **translateFromFrench** defined on it.

The code above is legal and does not cause any run-time errors. In order to see why the covariant redefinition in the first example causes problems while the one in the second example does not, details of product-type specifications have to be taken into account. In the first example, the translation will give the behavior **set_age** the type

$$\mathrm{glb}((\mathtt{T\_Person}, \mathtt{T\_Natural}) \rightarrow \mathtt{T\_Unit}, (\mathtt{T\_Child}, \mathtt{T\_SmallNatural}) \rightarrow \mathtt{T\_Unit})$$

which implies that the behavior application **set_age(aChild,n)** (where the type of **aChild** is **T_Child** and a type of **n** is **T_Natural**) is type-correct. This is so because the product-type of the argument is ⟨**T_Child**, **T_Natural**⟩, which is a subtype of ⟨**T_Person**, **T_Natural**⟩. However, which code should be executed when such a behavior application is encountered? Apparently, the code that just stores **n** in the appropriate slot of the object **aChild** would be type-incorrect. On the other hand, if code was provided that caps the natural argument to make it a small natural before storing it into the slot, the above behavior application would not cause the run-time error. In the presented type system, such an alternative code can be provided by using keyword := in the association. Therefore, it is not the *behavior specification* that is incorrect; rather it is the implicit assumption that the corresponding code directly stores the value into the slot. Such an implicit assumption is never made in the second example because it does not resemble structure definition as much as the first one does. In other words, it is not the type specification that fails here - it is our intuition.

Based on the above discussion, it is concluded that this problem can not and should not be dealt with at the *type and behavior specification* level, but only at the *code and structure definition* level. If the two levels (interface and structure definitions) were coupled as they are in many of today's programming languages, the novariance constraint would have to be enforced. However, in a type system that provides a clean separation between these two levels of abstraction (like the one presented here) the novariance restriction can be lifted at the higher level of behavior and type specification. This will be discussed further in Section 3.2.

In this section, mutable type specifications were discussed. A powerful user-defined assignment specification mechanism was introduced and its usage as well as its translation to product-type specifications were considered. The novariance condition usually placed on mutable types was also discussed; it was concluded that such a restriction is unnecessary at the type level for a type system that provides a clean separation between interface and structure definitions.

3.1.6 Parametric types and behaviors: specification

Parametricity is one of the most powerful mechanisms used for precise type specification. *Parametric polymorphism* that is present in type systems that support parametricity is comparable in power to *inclusion polymorphism* which is due to subtyping and interface inheritance.

The distinction between the two kinds of polymorphism is blurred in type systems with structural subtyping (BETA [MMPN93], Strongtalk [BG93], TM [BBdB$^+$93], ML [MTH90] and its clones). On the other hand, the designers of type-systems with user-defined (non-structural) subtyping usually focus on only one of the two forms of polymorphism, supporting the other one only minimally. For example, the type systems of Ada [Ada95] and Napier88 [MMPN93] focus primarily on parametric polymorphism, while the type systems of more traditional object-oriented languages (C++ [Str91] and clones, Java [AG96]) focus on inclusion polymorphism. Attempts to combine the full power of parametric and inclusion polymorphisms result in loss of substitutability, safety, or decidable typechecking (Java clones, Transframe [Sha97], Cecil [Cha93] and clones). The only type system that combines safety, substitutability, decidable typechecking, expressive parametric and inclusion polymorphism, and user-defined subtyping is $\mathrm{ML}_{\leq}$ [BM96a]. However, $\mathrm{ML}_{\leq}$ restricts user-defined subtyping to types within the same *type class* (a set of types with the same number of type arguments and the same kind of subtyping relationships between them), has a limited ability to deal with mutable types (like other ML clones), and does not provide separation between different specification levels (interface and implementation).

The type system presented in this dissertation is unique in that it combines all of the above properties. The parametricity interacts with all the components of the type system, making its consistent design a non-trivial task. Such interactions will be considered in this section along with examples showing the potential problems stemming from these interactions.

Consider the example of the specification of the type **T_List**:

```
type T_List {
    at(T_Natural) : T_???;
    cat(T_List) : T_List;
};
```

Here **at** extracts the n-th element from the list, while **cat** concatenates two lists. In type systems that do not support parametricity the type T_??? is usually specified to be **T_Object** (a supertype of all types). However, this leads to the *container problem* exemplified by the following code:

```
type T_Person {
    age() : T_Natural;
};
...
T_List list;
T_Person person1, person2, person3;
...
list := (person1, person2, person3);
list.at(1).age.print;
```

Here the last behavior application is considered to be illegal because the type-checker only has enough information to conclude that the result of **list.at(1)** will be **T_Object**, and the behavior **age** is not applicable to just objects, it is only applicable to persons. Thus, in this case the above type specification is too restrictive.

On the other hand, consider the code fragment

```
type T_Person {
    age() : T_Natural;
};
type T_Wine {
    age() : T_Natural;
};
...
T_List aWineList;
T_List aPersonList;
T_Person person1, person2, person3;
T_Wine wine1, wine2, wine3;
...
aPersonList := (person1, person2, person3);
aWineList := (wine1, wine2, wine3);
aPersonList := aPersonList.cat(aWineList);
```

The last assignment will store a mix of persons and wines in the variable **aPersonList** - a semantically incorrect action. However, the typechecker can not detect this incorrectness as its knowledge of the type of the variables **aPersonList** and **aWineList** is limited to the generic list type **T_List**. One can attempt to make the specification more strict by specifying types **T_PersonList** and **T_WineList** as unrelated subtypes of **T_List**:

```
type T_List {
    at(T_Natural) : T_Object;
    cat(T_List) : T_List;
};
type T_PersonList subtype of T_List {
    at(T_Natural) : T_Person;
    cat(T_PersonList) : T_PersonList;
};
type T_WineList subtype of T_List {
    at(T_Natural) : T_Wine;
    cat(T_WineList) : T_WineList;
};
...
T_PersonList aPersonList;
T_WineList aWineList;
T_Person person1, person2, person3;
T_Wine wine1, wine2, wine3;
...
aPersonList := (person1, person2, person3);
aWineList := (wine1, wine2, wine3);
aPersonList := aPersonList.cat(aWineList);
```

Then, the assignment in question will become type-incorrect, as the result of concatenation of a list of persons and a list of wines will have type `T_List`, which is not a subtype of `T_PersonList`. However, this approach requires a programmer to explicitly define list types for every possible element type – an activity which is both tedious and error-prone.

In order to provide static type safety in a clear and concise manner, the following *parametric type specification* can be utilized:

```
type T_List(X) {
    at(T_Natural) : X;
    cat(T_List(X)) : T_List(X);
};
```

The specification of the type `T_List` now has a *type parameter* `X`. The code in question will be

```
T_List(T_Person) aPersonList;
T_List(T_Wine) aWineList;
T_Person person1, person2, person3;
T_Wine wine1, wine2, wine3;
...
aPersonList := (person1, person2, person3);
aWineList := (wine1, wine2, wine3);
aPersonList := aPersonList.cat(aWineList);
```

and the last assignment will not type-check. Coming back to the container problem example, now the code

```
type T_Person {
    age() : T_Natural;
};
...
T_List(T_Person) list;
T_Person person1, person2, person3;
...
list := (person1, person2, person3);
list.at(1).age.print;
```

will typecheck, as the typechecker knows that the result type of `list.at(1)` is `T_Person` rather than just `T_Object`.

The number of parameters is not limited to one; for example, it is possible to specify a dictionary type as follows:

```
type T_MutableDictionary(KeyType,ValueType) {
    at(KeyType) :=: ValueType;
    allKeys() : T_List(KeyType);
    allValues() : T_List(ValueType);
    ...
};
```

and use it in code like the following:

```
T_MutableDictionary(T_String,T_Person) mdict;
T_Person p1, p2;
...
p1 := mdict.at("John");
...
mdict.at("Peter") := p2;
```

Parameters do not necessarily have to be types - they are only required to come from some lattice (a partially-ordered domain with lower and upper bounds). As an example, consider an array type that has a number of elements as one of its parameters:

```
type T_Array(ElementType, T_Natural UpperBound) {
    at(T_Natural) :=: ElementType;
}
```

This defines a parametric type `T_Array` with two parameters. The first parameter is called `ElementType`; it represents the type of array elements. The second one is called `UpperBound`; it represents the number of elements in the array. Since the second parameter is a natural number rather than a type, its specification includes the type specifier `T_Natural`.

In this section, the parametric type specifications were described. The next question related to parametric types is related to the specification of subtyping rules between them. Is the type of a list of students a subtype of the type of a list of persons? Can a parametric type be a subtype of a non-parametric one or vice-versa? The answers to these questions are given in the next section.

3.1.7 Parametricity and subtyping

Each parametric type specification defines a *family* of parametric types that differ only in their parameters. Therefore, the issue of subtyping parametric types can be divided into two: the issue of *intra-family* subtyping and that of *inter-family* subtyping.

Intra-family relationships

Intra-family subtyping is a subtyping relationship between types that belong to the same parametric type family. Consider an immutable list type specified as

```
type T_List(X) {
    at(T_Natural) : X;
    cat(T_List(X)) : T_List(X);
};
```

Let a type `T_Student` be a subtype of `T_Person`. Then, every student is also a person; therefore, a list of students can also be considered as a list of persons since each element of the former can be considered a person. In general, it is desirable to have a rule that specifies an intra-family subtyping relationship in terms of the relationships between parameter types, the same way it was specified for product (Equation 3.1) and behavior (Equation 3.2) types. In particular, for the immutable list types it is desirable to have the following rule:

$$(\mathtt{T\_List(X)} \preceq \mathtt{T\_List(Y)}) \overset{\text{def}}{\Longleftrightarrow} (\mathtt{X} \preceq \mathtt{Y})$$

I.e., the immutable list types are covariant in their first and only argument. This is specified as

```
type T_List(covar X) {
    at(T_Natural) : X;
    cat(T_List(X)) : T_List(X);
};
```

where the keyword **covar** is used to denote the *covariance* of the type parameter **X**.

Consider the following code:

```
T_List(T_Person) aPersonList;
T_List(T_Student) aStudentList;

aPersonList := aStudentList.cat(aPersonList);   // 1 Ok
aStudentList := aStudentList.cat(aPersonList);  // 2 Type error
aStudentList := aStudentList.cat(aStudentList); // 3 Ok
aPersonList := aStudentList.cat(aStudentList);  // 4 Ok
```

All these assignments are type-correct except for the second one. This is the expected result since a mix of students and persons can also be considered as a list of persons (every student is a person), while it cannot be considered a list of students (not every person is a student). On the other hand, a list of students can be considered to be a list of persons.

In order to see how the type-checker arrives at this conclusion, the behavior-product type translation of the above specification has to be considered:

```
type T_List(covar X);

const ((T_List(X), T_Natural) -> X) at;
const ((T_List(X), T_List(X)) -> T_List(X)) cat;
```

Thus, for a behavior application **cat(l1, l2)** to be valid, there should exist a type **X** such that

$$\langle \text{type of } \mathtt{l1}, \text{type of } \mathtt{l2} \rangle \preceq \langle \mathtt{T\_List(X)}, \mathtt{T\_List(X)} \rangle \tag{3.3}$$

and the result of this behavior application will then have type **T_List(X)**. Since the above statement is true for all types **X** that satisfy the above condition, the actual result type **T_R** has to be less than **T_List(X)** for all such **X**. This is equivalent to the statement that

$$\mathtt{T\_R} = \text{glb}\{\mathtt{X} \mid \mathtt{X} \text{ satisfies Equation 3.3}\}$$

Getting back to our example, the first behavior application **aStudentList.cat(aPersonList)** (or **cat(aStudentList, aPersonList)** in product notation) has argument product-type ⟨**T_List(T_Student)**, **T_List(T_Person)**⟩. Therefore the set of types **X** that satisfy Equation 3.3 is {**T_Person**}. The above is due to covariance of **T_List**: **T_List(T_Student)** $\preceq$ **T_List(T_Person)** because **T_Student** $\preceq$ **T_Person**. The greatest lower bound of this set is **T_Person**, and therefore the result type of this behavior application is **T_List(T_Person)**, which can be safely assigned to the variable of type **T_List(T_Person)** (the first assignment), but not to the variable of type **T_List(T_Student)** (the second assignment).

On the other hand, the set of types **X** that satisfy Equation 3.3 for the second behavior application **aStudentList.cat(aStudentList)** will be {**T_Person, T_Student**}; its greatest lower bound is **T_Student**, therefore the result type of this behavior application is **T_List(T_Student)** that can safely be assigned to a variable of type **T_List(T_Person)** (the third assignment) as well as to a variable of type **T_List(T_Student)** (the fourth assignment).

The above example dealt with a *covariant* type specification and an immutable list type. What happens when a *mutable* list type is considered? Such a type can no longer be covariant, since the resulting type specification would be unsound. If it was allowed, the following code would typecheck:

```
type T_Person;
type T_Student subtype of T_Person {
    studentId() : T_Natural;
};
type T_MutableList(covar X) {
    at(T_Natural) :=: X;
};

T_MutableList(T_Student) aStudentList;
T_Person aPerson;

aStudentList.at(1) := aPerson;  // Should be a type error!
aStudentList.at(1).studentId;
```

This code, however, will fail at run-time since the behavior **studentId** is not defined on persons.

The following is the reason why type-checking of the assignment succeeds: In product-behavior type notation, the above would read as

```
const ((T_MutableList(X), T_Natural) -> X) at;
const ((T_MutableList(X), T_Natural, X) -> T_Unit) set_at;

T_MutableList(T_Student) aStudentList;
T_Person aPerson;

set_at(aStudentList, 1, aPerson); // Should be a type error!
```

The set of types **X** such that

$$\begin{aligned}\text{argument type } &= \langle \mathtt{T\_MutableList(T\_Student)}, \mathtt{T\_Natural}, \mathtt{T\_Person}\rangle \\ &\preceq \langle \mathtt{T\_MutableList(X)}, \mathtt{T\_Natural}, \mathtt{X}\rangle = \text{ arguments for which } \mathtt{set\_at} \text{ is defined}\end{aligned}$$

will be $\{\mathtt{T\_Person}\}$. Indeed,

$$\langle \mathtt{T\_MutableList(T\_Student)}, \mathtt{T\_Natural}, \mathtt{T\_Person}\rangle \\ \preceq \langle \mathtt{T\_MutableList(T\_Person)}, \mathtt{T\_Natural}, \mathtt{T\_Person}\rangle$$

according to the subtyping rule for product types (Equation 3.1) and due to the covariance of list specification that gives

$$\mathtt{T\_MutableList(T\_Student)} \preceq \mathtt{T\_MutableList(T\_Person)} \tag{3.4}$$

This is the same situation as the one considered in Section 3.1.5: if the code that is written to implement **set_at** is just storing its argument into the array, it is not type-correct. In order to be able to write such code in a type-safe manner, the definition of the parametric type **T_MutableList** should be *novariant*:

```
type T_Person;
type T_Student subtype of T_Person {
    studentId() : T_Natural;
};
```

```
type T_MutableList(novar X) {
    at(T_Natural) :=: X;
};

T_MutableList(T_Student) aStudentList;
T_Person aPerson;

aStudentList.at(1) := aPerson;  // Type error
aStudentList.at(1).studentId
```

The parametric type **T_MutableList** is now defined as *novariant* (keyword **novar**), which means that none of the mutable list types is a subtype of the other, no matter which relationship exists between the type parameters of these list types.

The typechecking of this code fails because Equation 3.4 that led to successful typechecking of the above code in covariant mutable list specification is no longer true for novariant specification.

Why should a mutable list type be novariant? Informally, the type **T_MutableList(T_Student)** can not be a subtype of **T_MutableList(T_Person)** since the second one can accept into it any person, while the first one can not. On the other hand, the type **T_MutableList(T_Person)** can not be a subtype of **T_MutableList(T_Student)** because student's behaviors can be applied to an object extracted from the list of the second type while such behaviors can not be applied to an object extracted from the list of the first type.

So far, covariant and novariant specifications have been considered. The third possibility, a contravariant specification (keyword **contravar**), also exists. The following type specification uses contravariance to correctly type output streams:

```
type T_OutputStream(contravar X) {
    put() := X;
};
```

Note usage of the specifier := in the specification of behavior **put**: it means that this behavior can not be *accessed*, but only *assigned to*. Contravariance means that

$$(\texttt{T\_OutputStream(X)} \preceq \texttt{T\_OutputStream(Y)}) \overset{\text{def}}{\Longleftrightarrow} (\texttt{Y} \preceq \texttt{X})$$

(note the reversed direction of subtyping!).

Informally, both a student and a person can be put into an output stream of persons (a student is a person); however, only students can be put into an output stream of students. Therefore, an output stream of persons can do the same as an output stream of students (accept students) plus more (accept persons). Thus, the type of output streams of persons should be a subtype of the type of output streams of students.

Therefore, in the following code fragment

```
type T_Person;
type T_Student subtype of T_Person;

T_OutputStream(T_Student) aStudentOS;
T_OutputStream(T_Person) aPersonOS;
T_Person aPerson;
T_Student aStudent;

aPersonOS.put := aPerson;   // 1 Ok
aPersonOS.put := aStudent;  // 2 Ok
aStudentOS.put := aPerson;  // 3 Type error
aStudentOS.put := aStudent; // 4 Ok
```

all assignments are type-correct except for the third one, as it attempts to output a person to a stream designed for students.

In general, *read-only* entities (like immutable lists) are *covariant*, *write-only* entities (like output streams) are *contravariant*, and *read-and-write* entities (like mutable lists) are *novariant.*

Behavior types are examples of a parametric type family with two type parameters. The first of these parameters (argument, or input type) is contravariant, while the second one (result, or output type) is covariant (Equation 3.2). Thus, the behavior type discussed earlier in Section 3.1.4 can be specified as[11]

```
type ->(contravar ArgType, covar ResType) {
    apply(ArgType) : ResType;
};
```

This specification allows a programmer to use the behavior **apply** to apply behaviors to their arguments. This is type-safe behavioral reflection. It also shows how higher-order behaviors can be specified.

Product types considered in Section 3.1.3 can also be specified as parametric types in which all parameters are covariant (Equation 3.1).

Parametric types are also used for local variable type specification. The type implicitly used for local variables is a novariant type **T_Var** defined as follows:

```
type T_Var(novar X) {
    val() :=: X;
};
```

Whenever a local variable definition of the form **T_aType aName;** is encountered, it is translated into **const T_Var(T_aType) aName;**. Every assignment to this local variable of the form **aName := anExpression** is translated into **aName.val := anExpression**. Every other access to a local variable **aName** is translated into **aName.val**. The result is of course subject to normal product-type processing and expansion.

For example, the following code

```
type T_Person {
    age : T_Natural;
};
type T_Student subtype of T_Person;
...
T_Person aPerson;
T_Student aStudent;
...
aPerson := aStudent;
aPerson.age.print;
```

in fully expanded form becomes:

```
type T_Person;

const ->(T_Person, T_Natural) age;
type T_Student subtype of T_Person;
...
const T_Var(T_Person) aPerson;
const T_Var(T_Student) aStudent;
...
set_val(aPerson,aStudent);
print(age(val(aPerson)));
```

The typechecking algorithm is therefore free from any special treatment of local variables – they are simply regarded as objects of one of the **T_Var** types. The syntactic transformations described

[11]This is an implementation of *(APPLY)* test.

above do not allow the variables to "escape" from their scope, as there is no syntactic way to refer to a variable as an object. It is straightforward to verify that these transformations indeed give the intended meaning to the notion and typing of local variables. This will be formalized in Chapter 5.

In this section intra-family subtyping was considered. Concepts of covariance, contravariance, and novariance were discussed and examples of their use were presented. It has also been shown that intra-family subtyping is sufficient for definition of the fundamental types of the proposed type system, as well as for handling of local variables. This property demonstrates the power and reflexive capabilities of the presented type system.

Inter-family relationships

Another aspect of subtyping related to parametric types is *inter-family* subtyping. It occurs when a subtyping relationship is established between parametric types from different families.

Consider the following specification:

```
type T_Set(covar X) {
    isMember(X) : T_Boolean;
    union(T_Set(X)) : T_Set(X);
};
type T_List(covar X) subtype of T_Set(X) {
    at(T_Natural) : X;
};
```

It declares the type **T_List(X)** to be a subtype of **T_Set(X)** for every type **X**. For example, the following code is type-correct:

```
T_List(T_Student) aStudentList;
T_Set(T_Person) aPersonSet;

aPersonSet := aPersonSet.union(aStudentList);
```

The product-type argument type of the behavior application above is of type ⟨**T_Set(T_Person)**, **T_List(T_Student)**⟩. The minimal type **X** such that

$$\langle \texttt{T\_Set(T\_Person)}, \texttt{T\_List(T\_Student)} \rangle \preceq \langle \texttt{T\_Set(X)}, \texttt{T\_Set(X)} \rangle$$

(as required by the specification of the behavior **union**) is **X** = **T_Person**, since

$$\texttt{T\_List(T\_Student)} \preceq \texttt{T\_Set(T\_Student)} \preceq \texttt{T\_Set(T\_Person)} = \texttt{X}$$

Therefore, the result type of the behavior application above is **T_Set(T_Person)**, which is assignable to a variable of the same type.

In the above example, both types participating in inter-family subtyping relationship had the same number of type parameters with the same variance specification (**covar**). None of the above is required by the type system. For example, all of the following specifications are allowed:

```
type T_Set(covar X);
type T_EmptySet subtype of T_Set(X);

type T_List(covar X);
type T_String subtype of T_List(T_Character);
type T_UpdatableList(novar X) subtype of T_List(X);

type T_InputStream(covar X) {
    get : X;
};
type T_OutputStream(contravar X) {
        put := X;
    };
    type T_IOStream(novar X) subtype of T_InputStream(X), T_OutputStream(X);
```

These specifications define an empty set type that is a common subtype of all set types; a string type which is a subtype of the type of character lists; and an input-output stream type which is a subtype of appropriate input and output stream types.

An example streams type hierarchy for parameter types **T_Person** and its subtype **T_Student** is depicted in Figure 3.3. Note that this is an implementation of the test (*STREAMS*) from the test suite given in Section 2.6.

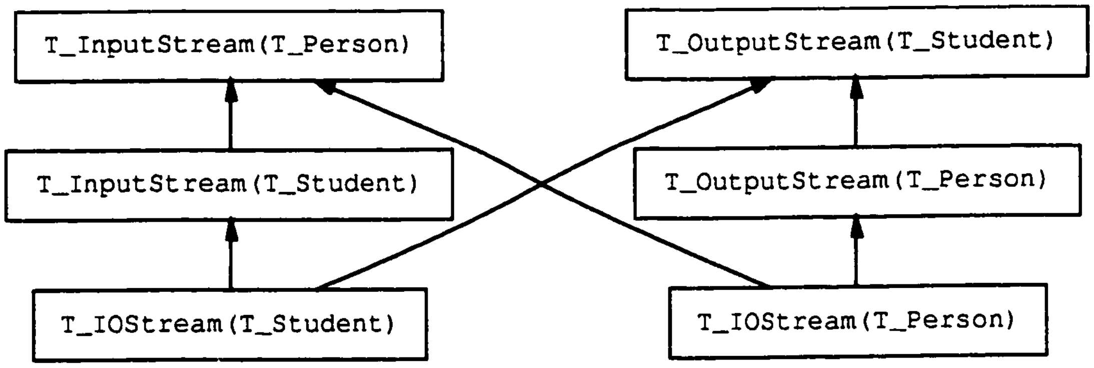

Figure 3.3: Stream types.

Not every set of (possibly parametric) subtype declarations results in a valid type system. For example, the type specification

```
    type T_MutableList(novar X) subtype of T_Strange;
    type T_Strange subtype of T_MutableList(T_Person);
```

is incorrect, since there exists a cycle

$$\texttt{T\_Strange} \preceq \texttt{T\_MutableList(T\_Person)} \preceq \texttt{T\_Strange}$$

In order to avoid situations like these, the following restriction is enforced. Let the *user type graph* $\mathcal{G}$ be defined as follows: for every parametric type family with constructor[12] **T_X**, there is a vertex **T_X** in $\mathcal{G}$. For every subtype specification of the form `T_X(...) <= T_Y(...)` there is a directed edge from **T_X** to **T_Y** in $\mathcal{G}$.

Rule 3.1 (Acyclicity). The user type graph $\mathcal{G}$ is acyclic. □

For example, the user type graph for the above type hierarchy will look as follows: **T_Strange** $\leftrightarrows$ **T_MutableList**. This graph clearly contains a cycle, and therefore this type hierarchy is rejected.

One of the applications of inter-family parametric type specifications is *F-bounded quantification* [CCH+89] which can be effectively used for specification of *binary methods*. Binary methods are behaviors with two arguments with identical types; their specification has always been a challenge for object-oriented type systems [BCC+96].

Consider the following situation (*COMPARABLE* test from the test suite given in Section 2.6). Assume it is necessary to specify types **T_Number** (with unrelated subtypes **T_Real** and **T_Radix**) and **T_Date** with comparison methods such that comparing two numbers or two dates between each other is legal, while their cross-comparison is not. A standard way of defining a type **T_Comparable** does not work because it is too permissive. For example, in the following code

[12] Simple types like **T_Person** are considered 0-ary constructors.

```
type T_Comparable {
    less(selftype) : T_Boolean;
};
type T_Number subtype of T_Comparable;
type T_Date subtype of T_Comparable;

T_Number n;
T_Date d;

n.less(d);        // Should have been a type error
```

the behavior application **n.less(d)** is type-correct. This is the case because the product argument type for this behavior application is ⟨**T_Number, T_Date**⟩ which is a subtype of ⟨**T_Comparable, T_Comparable**⟩ that specifies one of the argument product types where the behavior **less** is defined.

The problem can be solved by using F-bounded polymorphism [CCH+89] and parametricity. The following type specification

```
type T_Comparable(contravar X) {
    less(selftype) : T_Boolean;
};
type T_Number subtype of T_Comparable(T_Number);
type T_Real subtype of T_Number;
type T_Radix subtype of T_Number;
type T_Date subtype of T_Comparable(T_Date);
```

describes the necessary type hierarchy which is depicted in Figure 3.4.

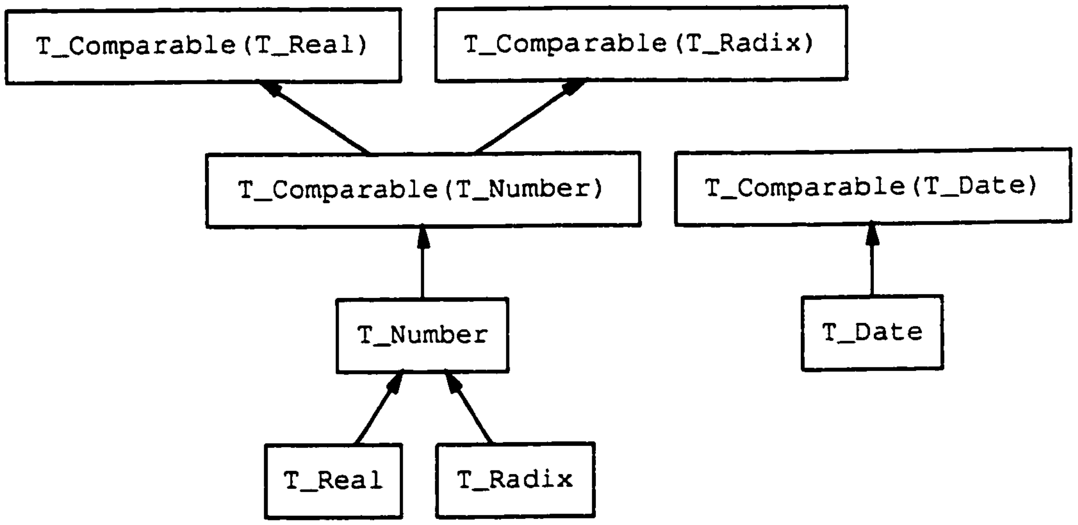

Figure 3.4: Comparable types.

Note that the type **T_Comparable** is *contravariant* in its only parameter. The following is the argument in favor of contravariance in this case: If an object of a type **T** can be compared to an object of type **X**, then it can also be compared to an object of any subtype **Y** of **X**. For example, since a real number (**T** = **T_Real**) can be compared to any number (**X** = **T_Number**), it can also be compared to a radix number (**Y** = **T_Radix**). Thus, for all types **T**, **X**, and **Y**

$$(\mathtt{T} \preceq \mathtt{T\_Comparable(X)} \wedge \mathtt{Y} \preceq \mathtt{X}) \Rightarrow \mathtt{T} \preceq \mathtt{T\_Comparable(Y)}$$

Let's put **T** = **T_Comparable(X)**. Then,

$$\forall \mathtt{X}, \mathtt{Y} : \mathtt{Y} \preceq \mathtt{X} \Rightarrow \mathtt{T\_Comparable(X)} \preceq \mathtt{T\_Comparable(Y)}$$

which is the definition of contravariance.

With the comparable type specification given above, the code

```
T_Number n;
T_Real re;
T_Radix ra;
T_Date d1, d2;

n.less(re);
re.less(ra);
ra.less(n);
d1.less(d2);
```

will be successfully typechecked, while the code

```
T_Number n;
T_Date d;

d.less(n);
```

will be rejected.

In this section, the interactions between parametricity and subtyping were considered. Both intra-family and inter-family relationships were discussed, and their power was illustrated by a number of examples. It was shown that the mechanisms described in this section are powerful enough to define behavior typing, variable typing, and F-bounded quantification. In the next section, the extension of the return type covariance rule discussed in Section 3.1.3 for the case of parametric types will be considered.

3.1.8 Parametricity and interface specification

In the presented type system, there are several rules that are imposed upon the user type specifications. One of them is the requirement of acyclicity of the user type graph $\mathcal{G}$ (Section 3.1.6); the other is the return type covariance rule (Section 3.1.3).

The return type covariance rule was formulated only for simple (non-parametric) types. A refinement of this rule for parametric types is required. Parametric specifications can introduce new problems related to return type covariance, as can be seen in the following code:

```
type T_Var(novar X) {
    val() :=: X;
};
type T_Object {
    makeRef() : T_Var(selftype);
};
type T_Person subtype of T_Object;

T_Object object;

// It is assumed that the dynamic type of aPerson is T_Person
// and the dynamic type of anObject is T_Object
object := aPerson;              // 1
object.makeRef().val := anObject; // 2 Dynamic type error
```

Here, both assignments are statically type-correct. However, the second assignment is dynamically incorrect since it tries to put **anObject** inside of a person reference created by the behavior application **object.makeRef()**. The latter creates a *person reference* rather than an *object reference* since the dynamic type of **object** is **T_Person**.

The problem here is related to the specification of the behavior **makeRef**. The expanded product-type definition of this behavior for types **T_Object** and **T_Person** yields

```
const (T_Object -> T_Var(T_Object)) makeRef;
const (T_Person -> T_Var(T_Person)) makeRef;
```

which contradicts the return type covariance rule[13]. The direct expansion used here is only done for illustrative purposes, since full expansion is in most cases infinite.

Therefore, the return type covariance rule has to be reformulated to deal with parametric specifications directly. The following is the desired formulation:

Rule 3.2 (Return type covariance). If $\{A_i(\alpha_1, \dots, \alpha_{n_i}) \rightarrow B_i(\alpha_1, \dots, \alpha_{n_i})\}_{i=1}^{n}$ are the specifications for a behavior b, then

$$\begin{aligned}
&\forall i\, \forall \alpha_1, \dots, \alpha_{n_i}, \alpha'_1, \dots, \alpha'_{n_i} : \\
&\quad A_i(\alpha_1, \dots, \alpha_{n_i}) \preceq A_i(\alpha'_1, \dots, \alpha'_{n_i}) \Rightarrow B_i(\alpha_1, \dots, \alpha_{n_i}) \preceq B_i(\alpha'_1, \dots, \alpha'_{n_i}) \\
&\forall i, j\, (i \neq j)\, \forall \alpha_1, \dots, \alpha_{n_i}, \alpha'_1, \dots, \alpha'_{n_j} : \\
&\quad A_i(\alpha_1, \dots, \alpha_{n_i}) \preceq A_j(\alpha'_1, \dots, \alpha'_{n_j}) \Rightarrow B_i(\alpha_1, \dots, \alpha_{n_i}) \preceq B_j(\alpha'_1, \dots, \alpha'_{n_j})
\end{aligned}$$

where $A(\alpha_1, \dots, \alpha_{n_i})$ denotes a parametric type with parameters $\alpha_1, \dots, \alpha_{n_i}$. □

This rule will be further refined in Chapter 4 (Definitions 4.5 and 4.10) to deal with constrained types.

The definition of the behavior **makeRef** does not satisfy Rule 3.2 since its product-type specification is $\alpha \rightarrow \texttt{T\_Var}(\alpha)$, and for $\alpha = \texttt{T\_Person}, \alpha' = \texttt{T\_Object}$ the first condition is violated:

$$A(\alpha) = \alpha = \texttt{T\_Person} \preceq \texttt{T\_Object} = \alpha' = A(\alpha')$$
$$B(\alpha) = \texttt{T\_Var}(\alpha) = \texttt{T\_Var(T\_Person)} \not\preceq \texttt{T\_Var(T\_Object)} = \texttt{T\_Var}(\alpha') = B(\alpha')$$

In ML [MTH90], the type constructor **ref** is defined similarly to the definition of **makeRef** and **T_Var** above (in ML, a type constructor plays a dual role: it is used for both type specification and run-time object creation). Therefore its usage causes problems. The ML solution to this problem is to treat this type constructor specially, severely restricting its polymorphism. In the presented type system, no special treatment is required for any type constructor and the effect of **ref** as a type constructor is accomplished using the mutable types considered in Section 3.1.5.

In this section, the extension of the return type covariance rule for the case of parametric specifications has been presented. Several examples illustrating the necessity and use of the extended rule have been considered.

3.1.9 Union and intersection types

One of the unique features of database programming languages is the incorporation of query facilities. Queries are declarative (side-effect free[14]) computations that are usually based on a variant of SQL. Queries require precise and adequate typing of set-theoretic and applicative operations, such as union, intersection, map etc.

With the help of parametric types, it is possible to define the behavior **union** on the type of immutable sets **T_Set** in such a way that it indeed has the desired properties:

```
type T_Set(covar X) {
    union(T_Set(X)): T_Set(X);
};
```

What happens if a union of a set of students and a set of persons is attempted? Such behavior application will have a product-type argument type ⟨**T_Set(T_Student)**, **T_Set(Person)**⟩, a product

[13] Indeed, T_Var(T_Person) is not a subtype of T_Var(T_Object) as T_Var is novariant.

[14] In object query models, a query may not necessarily be side-effect free; however, this does not affect query typing.

type in the specification of **union** is $\langle$`T_Set(X)`, `T_Set(X)`$\rangle$, and the greatest lower bound of types `X` such that

$$\langle \texttt{T\_Set(T\_Student)}, \texttt{T\_Set(Person)} \rangle \preceq \langle \texttt{T\_Set(X)}, \texttt{T\_Set(X)} \rangle$$

is `X` = **`T_Person`**. Therefore, the result type of the application will be **`T_Set(T_Person)`**. It is easy to show that the result type of an application of the behavior **union** to two sets with element types `A` and `B` will always be **`T_Set`**(lub(`A`, `B`)) — precise typing for the set-theoretic union operation.

The specification of behavior **map** is also possible using parametric and behavior types:

```
type T_Set(covar X) {
    map( (X):Y ): T_Set(Y);
};
```

The behavior **map** takes an argument which is a behavior applicable to the elements of the set and returns the set of results of applying this behavior, element-wise, to the elements of the source set. For example,

```
type T_Real {
    negate() : T_Real;
};

T_Set(T_Real) s;
s := (0.1, 0.2, 3.0);

s := s.map(negate);  // After the assignment, s will be (-0.1, -0.2, -3.0)
```

Only the specification of the behavior `intersect` causes problems. The result type of this behavior should be the greatest lower bound of the argument types because the resulting set contains only those objects that belong to the two argument sets at once. The presented type system provides a programmer with an ability to use type operators glb and lub explicitly in type specifications. Using these additional capabilities, the immutable set type can be specified as follows:

```
type T_Set(covar X) {
    union(T_Set(Y)) : T_Set(lub(X,Y));
    intersection(T_Set(Y)) : T_Set(glb(X,Y));
    map( (X):Y ) : T_Set(Y);
};
```

Note that the specification of **union** given here is equivalent to the specification given above. However, it describes the programmer's intentions more explicitly and is therefore preferable.

Using type specifiers glb and lub together with parametric and behavior types makes it possible to specify set-theoretic operations with an adequate precision. The mechanisms used here are not designed just for sets; they can be used for all types, including user-defined ones. This is in contrast to the approach taken in Machiavelli [BO96] and Fibonacci [AGO95], where precise typing of set-theoretic operations is also possible, but the treatment of sets is special and the programmer can not define new types with analogous typing precision.

3.1.10 Constrained specifications

The parametric type specifications considered so far were *unconstrained* in that any type could be used as their parameter. However, sometimes a parametric type only makes sense for some parameter types and not for all of them. In order to specify such conditions, *type constraints* are utilized.

The following is a specification of a type **`T_OrderedSet`** that requires that its elements be comparable with each other:

```
type T_OrderedSet(covar X)
  where (X subtype of T_Comparable(X))
  subtype of T_Set(X) {
    maximals() : T_Set(X);
    minimals() : T_Set(X);
};
```

This specification requires the type parameter X to be comparable. It also specifies that ordered sets are subtypes of sets. The behaviors **maximals** and **minimals** are defined to return sets of maximal and minimal elements of the set. Ordered sets are sets, and therefore all operations defined for sets are also applicable to them.

Type constraints can also be used to specify behaviors. For example, if the type of ordered sets is introduced just to define behaviors **maximals** and **minimals**, then it is possible to do without it and define the above behaviors on sets by adding the appropriate type constraints to their definitions:

```
type T_Set(covar X) {
    ...
    maximals() : T_Set(X) where X subtype of T_Comparable(X);
    minimals() : T_Set(X) where X subtype of T_Comparable(X);
};
```

This specification, however, does not allow one to declare variables of ordered set types, while the previous one did.

Constraints that can be used for type or behavior specification should not contain variables that are not used in the rest of the specification. They must also be *monotonic*. Monotonicity here means that type constraints must be constructed in a way that does not violate substitutability. The following is an example of how a non-monotonic constraint can result in a loss of type safety:

```
type T_Employed;
type T_CEO subtype of T_Employed;

type T_Company {
    fire(X) where (T_Employed subtype of X, X subtype of T_Employed); // *
};
...
T_Company c;
T_Employed e;
T_CEO ceo;
...
e := ceo;           // 1 Ok
c.fire(e);          // 2 Run-time error
```

The behavior **fire** is only applicable to employed persons that are not CEOs (the specification * says that the argument of this behavior must have type **T_Employed**, and **T_CEO** is not equal to **T_Employed**). The code above is statically type-correct (**e** is just an employed person, and so can be fired), but fails at run-time when an attempt is made to fire **ceo**. The reason for this failure is non-monotonicity of the constraint placed on the specification of the behavior **fire**: an employed person satisfies it, but a CEO does not, even though a CEO *is* an employed person.

In order to specify monotonicity requirements, a new translation step (besides the product-type translation) is introduced. The purpose of this step is to add any constraints implicitly specified for **selftype** to the constraint set. Every behavior specification that has an occurrence of **selftype** is translated as follows:

1. All occurrences of **selftype** are replaced by a fresh type variable S

2. A constraint S **subtype of** T(...) where T(...) is a type where the behavior is defined is added to the list of type constraints for that behavior

For every type T that defines a constraint that constraint is added to all behavior specifications where type T_T occurs.

For example, the full translation of the specification of the type of ordered sets (given above) will be:

```
type T_OrderedSet(covar X)
  where (X subtype of T_Comparable(X))
  subtype of T_Set(X);

const (T_OrderedSet(X) -> T_Set(X)
            where X subtype of T_Comparable(X)) maximals;
const (T_OrderedSet(X) -> T_Set(X)
            where X subtype of T_Comparable(X)) minimals;
```

while the translation of the type specification of real numbers' type

```
type T_Real {
    add(selftype) : selftype;
    subtract(selftype) : selftype;
    multiply(selftype) : selftype;
    divide(selftype) : T_Real;
    negate() : selftype;
};
```

will be

```
type T_Real;
const ((S, S) -> S where S subtype of T_Real) add;
const ((S, S) -> S where S subtype of T_Real) subtract;
const ((S, S) -> S where S subtype of T_Real) multiply;
const ((S, S) -> T_Real where S subtype of T_Real) divide;
const (S -> S where S subtype of T_Real) negate;
```

Now the monotonicity requirement can be defined. It has two parts: one deals with type specification constraints, and the other with constraints placed on behaviors. If a set of constraints $C(\alpha_1, \ldots, \alpha_n)$ is placed on a type specification of a type $T(\alpha_1, \ldots, \alpha_n)$, then monotonicity requires that

$$\forall \alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_n : C(\alpha'_1, \ldots, \alpha'_n) \wedge (T(\alpha_1, \ldots, \alpha_n) \preceq T(\alpha'_1, \ldots, \alpha'_n)) \Rightarrow C(\alpha_1, \ldots, \alpha_n)$$

Thus, the constraint placed on the specification of T_OrderedSet is monotonic, as

$$\begin{aligned}\forall \alpha, \alpha' : \ & \alpha' \preceq \texttt{T\_Comparable}(\alpha') \wedge \texttt{T\_OrderedSet}(\alpha) \preceq \texttt{T\_OrderedSet}(\alpha') \\ & \Rightarrow \alpha \preceq \alpha' \preceq \texttt{T\_Comparable}(\alpha') \preceq \texttt{T\_Comparable}(\alpha) \\ & \Rightarrow \alpha \preceq \texttt{T\_Comparable}(\alpha)\end{aligned}$$

due to covariance of T_OrderedSet and contravariance of T_Comparable.

If a set of constraints $C(\alpha_1, \ldots, \alpha_n)$ is placed on a behavior specification

$$A(\alpha_1, \ldots, \alpha_n) \rightarrow R(\alpha_1, \ldots, \alpha_n)$$

then monotonicity requires that

$$\forall \alpha_1, \ldots, \alpha_n, \alpha'_1, \ldots, \alpha'_n : C(\alpha'_1, \ldots, \alpha'_n) \wedge A(\alpha_1, \ldots, \alpha_n) \preceq A(\alpha'_1, \ldots, \alpha'_n) \Rightarrow C(\alpha_1, \ldots, \alpha_n)$$

Thus, the constraint placed on the behavior specification of maximals is monotonic (the same proof as for the previous example), while the one placed on fire is not.

These monotonicity conditions will be given a more formal treatment in Chapter 5.

In this section, constraints and constrained specifications were considered. The monotonicity conditions that the constraints must satisfy have been formulated and illustrated.

3.1.11 Subtyping and inheritance

While subtyping is a very powerful concept, sometimes it is too restrictive. There are cases when it is necessary to reuse an interface without creating a subtype. An example of such situation is demonstrated by the *LIST* test from Section 2.6, where a single linked list node type and a double linked list node type have to be specified.

The straightforward approach to this problem

```
type T_LinkedListNode {
    getNext() : selftype;
    attach(selftype);
};
type T_DoubleLinkedListNode subtype of T_LinkedListNode {
    getPrev() : selftype;
    attachLeft(selftype);
};
```

is unsatisfactory because it allows nodes of different types to be mixed in a single list. This is so because **T_DoubleLinkedListNode** is specified as a subtype of **T_LinkedListNode** and can therefore be used everywhere a linked list node can be used, including a single linked list. If a programmer does not want to allow this behavior, the above specification will not do. In this case, F-bounded polymorphism [CCH⁺89] can be used. This solution is analogous to the solution of the *COMPARABLE* problem presented in Section 3.1.7.

```
type T_LinkedListNodeInterface(novar X) {
    getNext() : X;
    attach(X);
};
type T_LinkedListNode subtype of T_LinkedListNodeInterface(T_LinkedListNode);
type T_DoubleLinkedListNode
  subtype of T_LinkedListNodeInterface(T_DoubleLinkedListNode) {
    getPrev() : T_DoubleLinkedListNode;
    attachLeft(T_DoubleLinkedListNode);
};
```

This solution is type-correct and gives the intended type semantics to both node types. However, it is not nearly as elegant as the solution based on *matching* [BFP96]. Another way of representing this situation will be given in Section 3.2.2, where *class extension* rather than subtyping will be utilized.

3.1.12 Conclusions

In this section, the type and behavior specifications were described. The use of subtyping, product types, behavior types, parametric types, and constrained types was illustrated and discussed. Several transformation techniques used to translate the user-defined specifications into a more basic, primitive notation were introduced. Examples considered in this section show that the presented type system is capable of typing all tests[15] from Section 2.6 without sacrificing substitutability, static type safety, and decidable typechecking (as will be shown in Chapter 5).

Type and behavior specifications form the most abstract layer of the proposed type system, since they deal only with interfaces, ignoring object structure, the code implementing behaviors, and underlying physical memory layout. The middle layer of the presented type system consists of *classes* that describe object structure and *functions* that implement behaviors. This layer will be considered in the next section.

[15]Except for the (*BROWSER*) test that will be discussed in the next section.

3.2 Classes and functions

Interfaces play an important role in object-oriented programming. Yet, an interface specification is insufficient for programming since both object structure and code implementing the interfaces for a particular structure has to be specified before a program can be successfully translated (compiled or interpreted). The intermediate layer of abstraction which consists of *classes* (structure) and *function* (code) specifications is designed for this purpose.

In this section, classes and functions will be described. Issues related to dispatch will also be discussed. Connections between the intermediate layer of abstraction and the higher (interface) layer, such as correspondence between classes and types, behaviors and functions, and typechecking issues will also be elaborated.

3.2.1 Classes

While types denote concepts at a very abstract level, *classes* give a more concrete structural view of objects used in a program. Every object belongs to a class, as no material object can exist without a structure. A type may be implemented by a single class, or by several different classes, or it may not be implemented at all (an *abstract type*). *Classes* specify object structure at an intermediate level: they are not as abstract as types, yet they are abstract enough not to specify a particular physical memory layout for the objects that belong to the class. The following is an example class specification:

```
type T_Person {
    age :=: T_Natural;
    name :=: T_String;
};
class C_Person implements T_Person {
    T_Natural _age;
    T_String _name;
    age implementation _age;
    name implementation _name;
};
```

The class **C_Person** implements the type **T_Person**; the behavior **age** is implemented by the *attribute* **_age**[16] and the behavior **name** is implemented by the attribute **_name**. This implementation of the type **T_Person** is rather trivial, while the next example shows a non-trivial implementation of the same type:

```
class C_Person3Names implements T_Person {
    T_Natural _age;
    T_String _firstName;
    T_String _secondName;
    T_String _middleName;
    age implementation _age;
    name : T_String implementation fun() {
       return _firstName + ' ' + _middleName + ' ' + _secondName;
    };
    name := T_String implementation fun(arg) {
        _firstName := arg.columnSeparatedBySpaces(1);
        _middleName := arg.columnSeparatedBySpaces(2);
        _secondName := arg.columnSeparatedBySpaces(3);
    };
};
```

[16]The notation **_attr** will be used for attributes to distinguish them from behaviors.

Here, a person is implemented by a class with four attributes. The first one corresponds to a person's age, while the other three represent different components of a person's name. Therefore, the behavior **name** can no longer be implemented as a simple attribute access; instead, it is implemented as a pair of anonymous *functions* whose bodies are written in curly braces. Two functions are necessary because the behavior **name** was specified as *assignable* (token `:=:`). One function is used to "access" the name, and the other is used to "set" it.

It is perfectly legal for a program to have both **C_Person** and **C_Person3Names** implementing the same type **T_Person**. Objects of both classes can exist simultaneously in the program. Only the programmer of the person classes needs to know about them. Users only need to see the type. For example, the following code

```
T_Person person;
...
person.name := "John A Smith";
...
person.age.print;
person.name.print;
```

will work for objects of both person classes. Moreover, it will also work for *any* class that implements the type **T_Person**.

The ability of the code above to work correctly for all possible implementations of **T_Person** is a consequence of using just *types* and not *classes* for variable and argument type specifications. Since the code never refers to an actual object structure (class), only the interface specified by a type is used for typechecking. Successfully typechecked code therefore relies on *interface* only and does not need to be changed if the object structure changes.

This property is very important for code reuse in evolving systems, such as database applications where the legacy code constitutes a significant portion of overall application code. When data structure evolves, the legacy code is still valid and does not have to be rewritten.

The only place where classes are implicitly used for argument specification is the code of functions written for a particular class. The functions implementing behaviors in a class need to access attributes of that class. Therefore, the implicit first argument of those functions (the receiver) is implicitly typed by the enclosing class. For the purposes of typechecking, a class translates to a special kind of type, while an attribute translates into a pair of behaviors, one for getting and the other for setting the value of the attribute being translated.

For example, the specification of the class **C_Person3Names** given above will be translated into

```
type T_Person {
    age : T_Natural;
    set_age(T_Natural);
    name : T_String;
    set_name(T_String);
};
type C_Person3Names subtype of T_Person {
    // translation of the attribute _age
    _age : T_Natural implementation fun() <system-defined code>;
    set__age(T_Natural) implementation fun(arg) <system-defined code>;
    // translation of the attribute _firstName
    _firstName : T_String implementation fun() <system-defined code>;
    set__firstName(T_String) implementation fun(arg) <system-defined code>;
    // translation of the attribute _secondName
    _secondName : T_String implementation fun() <system-defined code>;
    set__secondName(T_String) implementation fun(arg) <system-defined code>;
    // translation of the attribute _middleName
    _middleName : T_String implementation fun() <system-defined code>;
    set__middleName(T_String) implementation fun(arg) <system-defined code>;

    age implementation fun() { return _age; };
    set_age(T_Natural) implementation fun(x) { _age := x; };
```

```
        name implementation fun() {
           return _firstName + ' ' + _middleName + ' ' + _secondName;
        };
        set_name(T_Natural) implementation fun(arg) {
            _firstName := (arg.columnSeparatedBySpaces(1));
            _middleName := (arg.columnSeparatedBySpaces(2));
            _secondName := (arg.columnSeparatedBySpaces(3));
        };
    };
```

after class and assignment expansion. The **implementation** clauses are the *association specifications*: they specify the code (function) that is associated to a particular behavior on arguments of a particular type. Functions, associations, and issues related to their consistency will be discussed in Section 3.2.4.

In the above example it has been shown how the class specifications can be translated into type specifications for type checking purposes. It also shows that classes (**C_Person3Names**) are translated into subtypes of the types they implement (**T_Person**), so an object of such a class can be used everywhere its type is required.

While a type can be implemented by several classes, a class always implements a single type. A family of parametric types can be implemented by a family of parametric classes. For example, the following specifies the family of classes **C_Set(X)** that implements the family of parametric types **T_Set(X)**:

```
    type T_Set(covar X) {
        isElement(X) : T_Boolean;
        union(T_Set(X)) : T_Set(X);
    };
    class C_Set(X) implements T_Set(X) {
        const T_List(X) _list;
        ...
    };
```

The set is implemented as a list. Since sets are immutable, the attribute **_list** was specified as constant (specifier **const**) thus disallowing assignment to the attribute. This specification allows for extensive reuse since the attribute **_list** is specified to have a *type* **T_List(X)** and will thus work with any possible implementation of a list.

The parametric class specification is not as elaborate as the parametric type specification, as each class has to correspond to a single type. Thus, the following class specification is illegal:

```
    class C_Set implements T_Set(X);
        ...
```

since it would make **C_Set** correspond to *all* types **T_Set(X)**. The above restriction is important since classes are responsible for *object creation*. If a class could correspond to several types, the type of an object created via that class would be ambiguous.

Another way of parameterizing a class is by explicit use of **selftype**. For example, the following is a specification of a linked list node class:

```
    type T_LinkedListNode {
        getNext() : selftype;
        attach(selftype);
    };
    class C_LinkedListNode implements T_LinkedListNode {
        selftype _next;
        getNext() : selftype implementation _next;
        attach(selftype) implementation ...
        ...
    };
```

The resulting class **C_LinkedListNode** has a mutable attribute **_next** that is typed by **selftype**. The reason for the use of **selftype** instead of plain **T_LinkedListNode** will be apparent when the notions of *subclassing* and *class extension* are introduced in the next section.

3.2.2 Subclassing and class extension

Just as *subtyping* is used to achieve inclusion polymorphism when types are concerned, *subclassing* can be used to achieve the same effect for classes. Subclassing as well as subtyping implies substitutability, and the subclassing semantics is the semantics of subtyping given by the translation of classes to types introduced in the previous section. For example, the following is a specification of the class **C_Student** as a subclass of **C_Person**:

```
type T_Person {
    age :=: T_Natural;
    name :=: T_String;
};
class C_Person implements T_Person {
    T_Natural _age;
    T_String _name;
    age implementation _age;
    name implementation _name;
};
type T_Student subtype of T_Person {
    studentId :=: T_Natural;
};
class C_Student subclass of C_Person implements T_Student {
    T_Natural _studentId;
    studentId implementation _studentId;
};
```

The class **C_Student** has the same attributes as the person class **C_Person** as well as the additional attribute **_studentId**. In this case, the type **T_Student** which is a subtype of **T_Person** is implemented by a subclass **C_Student** of the class **C_Person**. This relationship is neither necessary nor required. For example, the following specifies unrelated classes for the same two types:

```
type T_Person {
    age :=: T_Natural;
    name :=: T_String;
};
class C_Person implements T_Person {
    T_Natural _age;
    T_String _name;
    age implementation _age;
    name implementation _name;
};
type T_Student subtype of T_Person {
    studentId :=: T_Natural;
};
class C_Student3Names implements T_Student {
    T_Natural _age;
    T_String _firstName;
    T_String _secondName;
    T_String _middleName;
    T_Natural _studentId;
    age implementation _age;
```

```
        name : T_Natural implementation fun() {
           return _firstName + ' ' + _middleName + ' ' + _secondName;
        };
        name := T_Natural implementation fun(arg) {
            _firstName := arg.columnSeparatedBySpaces(1);
            _middleName := arg.columnSeparatedBySpaces(2);
            _secondName := arg.columnSeparatedBySpaces(3);
        };
        studentId implementation _studentId;
    };
```

Thus, subtyping does not imply subclassing, but subclassing does imply subtyping.

This definition of subclassing is parallel to the definition of subtyping. Traditionally, however, the term *subclassing* denotes the notion of *class extension*, in which only the structure is inherited and no subtyping or substitutability is required. In the presented type system *class extension* is a mechanism for such structure reuse. It does not imply substitutability or subtyping. The need for a relationship that is weaker than subtyping stems from the fact that mutable attributes are novariant (Section 3.1.5). For example, the following specification is incorrect:

```
    type T_Person {
        age :=: T_Natural;
        name :=: T_String;
    };
    class C_Person implements T_Person {
        T_Natural _age;
        T_String _name;
        ...
    };
    type T_Child {
        age :=: T_SmallNatural;
        name :=: T_String;
        favoriteToys : T_Set(T_Toy);
    };
    class C_Child subclass of C_Person implements T_Child {
        T_SmallNatural _age;
        T_Set(T_Toy) _ftoys;
        ...
    };
```

since the class **C_Child** redefines mutable attribute **_age** and therefore can not be used everywhere the class **C_Person** is used.

However, the following specification is allowed:

```
    type T_Person {
        age :=: T_Natural;
        name :=: T_String;
    };
    class C_Person implements T_Person {
        T_Natural _age;
        T_String _name;
        ...
    };
    type T_Child {
        age :=: T_SmallNatural;
        name :=: T_String;
        favoriteToys : T_Set(T_Toy);
```

```
};
class C_Child extends C_Person implements T_Child {
    T_SmallNatural _age;
    T_Set(T_Toy) _ftoys;
    ...
};
```

In this specification, the class **C_Child** *extends* the class **C_Person** rather than *subclasses* it. Therefore it is possible to redefine the attribute **_age** while inheriting the attribute **_name** with no modification.

In the presented type system, class extension is understood as textual rewriting of the class with automatic substitution of explicit and implicit occurrences of **selftype** in the style of [PS94]. The definitions given in the extending class override the definitions of the class being extended (just like the definition of the attribute **_age** in the above example). A class can extend several classes, and the conflicts have to be explicitly resolved by the programmer. Since class extension does not imply subtyping, the code inherited from the class being extended needs to be retypechecked in the context of the extending class.

The following is the example of usage of class extension along with the **selftype** mechanism:

```
type T_LinkedListNode {
    getNext() : selftype;
    attach(selftype);
};
class C_LinkedListNode implements T_LinkedListNode {
    selftype _next;
    getNext() : selftype implementation _next;
    attach(selftype) implementation ...
};

type T_DoubleLinkedListNode {
    getNext() : selftype;
    getPrev() : selftype;
    attach(selftype);
    attachLeft(selftype);
};
class C_DoubleLinkedListNode
        extends C_LinkedListNode
        implements T_DoubleLinkedListNode {
    selftype _prev;
    getPrev() implementation _prev;
    attachLeft(selftype) implementation ...
};
```

Even though the type **T_DoubleLinkedListNode** is not a subtype of **T_LinkedListNode** and the class **C_DoubleLinkedListNode** is not a subclass of **C_LinkedListNode**, it is still possible to reuse the code and specification of the former in the specification of the latter. The programmer does not need to rewrite the code if a subtype can not be produced, since class extension can be used to avoid code duplication (the implementations of behaviors **attach** and **next** and the attribute **_next** are inherited from **C_LinkedNodeType**). Note that this is another solution of the (LIST) test (Section 2.6), which is in many ways more elegant and concise than the one given in Section 3.1.11.

Class extension need not be parallel to subtyping; in fact, they can legally go in opposite directions. Consider the specification of circles and ellipses. A circle *is-a* ellipse, and therefore its type should be a subtype of that of an ellipse. From the structural point of view, however, a circle can be represented by center and radius (major axis), while for an ellipse center, major axis, and minor axis are needed[17]. Therefore, the structure of an ellipse is an extension of the structure of a circle.

[17]We assume that the major axis is always parallel to X-axis. Otherwise, an additional attribute would be required.

This situation is extremely difficult to model in most of the today's type systems and languages[18]. Yet, it can be elegantly represented in the type system described here:

```
type T_Ellipse {
    center() :=: T_Point;
    majorAxis() :=: T_Length;
    minorAxis() :=: T_Length;
};
type T_Circle subtype of T_Ellipse;

class C_Circle implements T_Circle {
    T_Point _center;
    T_Length _majorAxis;
    center implementation _center;
    majorAxis implementation _majorAxis;
    minorAxis implementation _majorAxis;
};
class C_Ellipse extends C_Circle implements T_Ellipse {
    T_Length _minorAxis;
    minorAxis implementation _minorAxis;
};
```

Note how the same attribute **_majorAxis** is used in the class **C_Circle** to implement both **majorAxis** and **minorAxis** behaviors and how the implementation of **minorAxis** is overridden in **C_Ellipse** to refer to the new attribute **_minorAxis** instead. This specification allows circles to be used everywhere ellipses can be used, yet it allows the structure of circles to be reused for ellipses.

In this section, both subclassing and class extension have been described. Their use was illustrated by several examples. It has also been shown how class specifications are transformed into type definitions for typechecking purposes.

3.2.3 Object creation and extent maintenance

Classes not only define the object structure, they are also used to create objects. In fact, objects can *only* be created via classes. Since classes themselves are objects, the object creation is done by applying the creation behavior **new** to the class object. There are two differences between object creation and regular behavior application. First, the creation behavior is predefined and does not have any arguments. Second, when a class is explicitly used, the *constructor expression* can be used to initialize its attributes, both mutable and immutable. This is important as only constructor expressions can put values into immutable attributes. Consider the following example where immutable circles and ellipses are defined:

```
type T_Ellipse {
    center() : T_Point;
    majorAxis() : T_Length;
    minorAxis() : T_Length;
};
type T_Circle subtype of T_Ellipse;

class C_Circle implements T_Circle {
    const T_Point _center;
    const T_Length _majorAxis;
    center implementation _center;
    majorAxis implementation _majorAxis;
```

[18] As a C++ programmer, the author has encountered numerous situations of this kind. Every time, either type safety or code reuse had to be compromised.

```
        minorAxis implementation _majorAxis;
    };
    class C_Ellipse extends C_Circle implements T_Ellipse {
        const T_Length _minorAxis;
        minorAxis implementation _minorAxis;
    };
```

Then, the following would create a circle with a unit radius:

```
    T_Circle aCircle := new C_Circle(
        _center = new C_Point (...),
        _majorAxis = 1.0
        );
```

The inability to redefine behavior **new** is not nearly as serious a restriction as it seems to be. Consider the following specification of a person type and class:

```
    type T_Person {
        age :=: T_Natural;
        name :=: T_String;
    };

    class C_Person implements T_Person {
        T_Natural _age;
        T_String _name;
        name implementation _name;
        age implementation _age;
        initialize(T_Natural initAge, T_String initName) implementation {
                if (initName.isEmpty or initAge.isEmpty) then {
                    raise incorrectInit;
                };
                _age := initAge;
                _name := initName;
            };
        ...
    };
    // Create a new person
    T_Person aPerson := (new C_Person).initialize(25, "John Smith");
```

Note that the behavior **initialize** is defined in the class **C_Person** rather than in its type. This means that the behavior **initialize** can only be legally used on objects whose class is statically known to be **C_Person**. Since classes can not be explicitly used for any specification, the only time the class is statically known is when the object has just been created, as in the example above.

Classes are also used for extent maintenance. Most of the classes maintain a collection of all their objects (the *extent*) automatically. The extent can be used to query over all objects of a specified class. The behavior **extent** is defined on classes to return a set of their objects. It can be used as follows:

```
    // Print all persons created so far
    C_Person.extent.foreach(print);
```

Not all classes allow object creation or support object maintenance. In fact, there is a taxonomy of classes. *Regular* classes support object creation and maintain their extents. *Finite* classes support extents, but do not allow object creation (classes of characters and booleans are examples of finite classes). *Manual* classes allow object creation, but do not automatically support their extents (e.g. classes of lists and sets). Finally, *infinite* classes support neither extent nor object creation (such as classes of real and integer numbers).

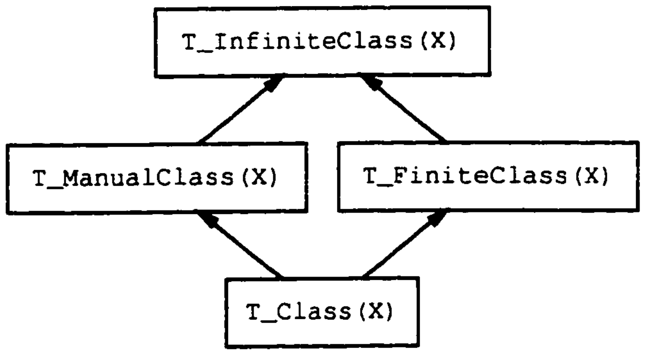

Figure 3.5: Class type hierarchy

Since classes are objects, the above taxonomy can be neatly expressed in terms of the type hierarchy of classes:

```
type T_InfiniteClass(covar X);
type T_FiniteClass(covar X) subtype of T_InfiniteClass(X) {
    extent() : T_Set(X);
};
type T_ManualClass(covar X) subtype of T_InfiniteClass(X) {
    new() : X;
};
type T_Class(covar X) subtype of T_ManualClass(X), T_FiniteClass(X);
```

When a class is specified, an appropriate object of one of the above types is created. The class type hierarchy is depicted in Figure 3.5.

Classes represent an intermediate layer of structure specification. Their functional counterparts at the intermediate layer are *functions* considered in the next section.

3.2.4 Functions and associations

Behaviors are abstract entities. They describe functionality at the highest level, via signatures. For the actual computation, however, signatures can not be used and therefore more concrete functionality specification has to be provided. In the presented type system, *functions* play this role. A *function* defines functionality via a code fragment that is executed when the function is invoked.

The code that is used in functions is composed of a sequence of behavior applications. Inside the code, local variable declarations and certain control statements are allowed. For the purposes of typechecking, only assignment, **return** and **typeif** (discussed later) statements are of interest. The following is an example function:

```
fun(x) {
    T_Integer result;
    result := x.age - age;
    return result;
};
```

This function can not be typechecked since it specifies neither its argument nor return type. However, functions do not exist by themselves[19]. Before a function can be used, it has to be *associated* with a particular behavior on a set of argument types. This is an example association:

```
type T_Person {
   age :=: T_Natural;
```

[19]Except for *closures* discussed later.

```
    ageDifference(T_Person x) : T_Integer implementation
      fun(x)   // <- This line can be omitted, as it can be
               // inferred from the behavior signature
      {
         T_Integer result;
         result := x.age - age;
         return result;
      };
};
```

This association can be typechecked to determine whether the function *conforms* to the signature of the behavior it is associated with. The issues of typechecking are discussed in Chapter 4, and its formal underpinnings in Chapter 5. An informal overview of typechecking and type system capabilities was already given in Section 3.1, therefore in this section only issues specific to function associations, dispatch, and special statements will be discussed.

A function can also be associated with a behavior on a *class* instead of a type. Such a function has access to class attributes, as in the following example:

```
type T_Person {
    age :=: T_Natural;
    name :=: T_String;
};
class C_Person3Names implements T_Person {
    T_Natural _age;
    T_String _firstName;
    T_String _secondName;
    T_String _middleName;
    age implementation _age;
    name : T_String implementation fun() {
       return _firstName + ' ' + _middleName + ' ' + _secondName;
    };
    name := T_String implementation fun(arg) {
        _firstName := arg.columnSeparatedBySpaces(1);
        _middleName := arg.columnSeparatedBySpaces(2);
        _secondName := arg.columnSeparatedBySpaces(3);
    };
};
```

Here the functions that implement the two (set and get) branches of the behavior **name** are associated with these branches on the class **C_Person3Names**.

Note that the ability to associate functions with both types and classes is an extension of a standard object-oriented model. For example, Java [AG96] supports separation between types (interfaces) and classes. However, it only allows class associations to be made, while providing single subclassing. This is too restrictive for many practical purposes. Consider the specification of a printer, a fax machine, a copier, and an all-in-one machine. The first three devices know how to print, fax, or copy a document, while the last device combines the properties of the three. Since the methods for printing, copying, and faxing are the same for the corresponding devices and the all-in-one machine, it is desirable to reuse the functions written for the type of the latter. The following is the required specification:

```
type T_Printer {
    print(T_Document) implementation ...
};
type T_Copier {
    copy(T_Document) implementation ...
};
```

```
type T_Fax {
    fax(T_Document) implementation ...
};
type T_AllInOne subtype of T_Printer, T_Copier, T_Fax;
```

In this example, the type T_AllInOne inherits both the behavior specifications and their implementations from its three parent types. Such an arrangement would be impossible in Java, and all the code would have to be repeated twice.

What does it mean for a function to be *inherited*? It means that the function code can not only be used in the type it is specified on, but also in subtypes that do not override the behavior association for that function. The following example illustrates this point (another example of code inheritance was given in Section 3.1.3):

```
type T_Account {
    withdraw(T_Amount amount) implementation  ... // a
type T_ChequingAccount subtype of T_Account;
type T_RRSPAccount subtype of T_Account {
    withdraw(T_Amount amount) implementation ... // b
};
...
T_Account anAccount;
T_ChequingAccount aChequingAccount;
T_RRSPAccount anRRSPAccount;
...
anAccount.withdraw(100);          // 1
aChequingAccount.withdraw(100);   // 2
anRRSPAccount.withdraw(100);      // 3
```

A generic account allows a person to withdraw money and provides an implementation (function a) for the behavior withdraw. The same implementation can be used for chequing accounts, since the withdrawal procedure is the same. However, registered retirement savings plan accounts have a totally different withdrawal procedure implemented by the function b. Therefore, the behavior applications 1 and 2 will dispatch to function a, while the behavior application 3 will dispatch to function b. The same code is used for types T_Account and T_ChequingAccount.

3.2.5 Dispatch

The process of choosing the function to execute according to the receiver type is called *dispatch*. In the presented type system, multiple dispatch (Section 3.1.3) is adopted.

Multiple dispatch is a complicated process. The description of dispatch algorithms used for multiple dispatch is outside the scope of this dissertation. Some of the proposed algorithms can be found in [DCG94], [DGC95], [CTK94], [CT95], and [DAS96]. However, in a typechecked program, dispatch should never end with a "method not understood" or "message ambiguous" error. The conditions that the specifications are required to meet in order to provide such an assurance are outlined below.

Some terminology first. The association of the form

```
<behavior>(<product-type>) implementation <function>
```

defines that a function <function> is associated with the behavior <behavior> on types that participate in the <product-type>. For example, the following

```
compare(T_Point, T_Point) implementation fun(point1, point2) {
        return point1.x == point2.x and point1.y == point2.y ;
    };
```

denotes association between the behavior **compare** and the function given on the product type (**T_Point, T_Point**). In other words, it is assumed that if two points are arguments to the behavior application of **compare**, the above function should execute.

A behavior can have several associations, for different argument types. In the presented system it is required that the product-types that participate in the associations be different up to parametric type families. In other words, the following associations can coexist:

```
isEmpty(T_List(X)) implementation <function1>;
isEmpty(T_Set(X)) implementation <function2>;
```

while the following ones can not:

```
isEmpty(T_List(Person)) implementation <function1>;
isEmpty(T_List(String)) implementation <function2>;
```

since in the second case the argument types are from the same parametric family (**T_List**) and are not "distinct enough".

The reason for the above restriction is the multi-fold increase in dispatch complexity in case the restriction is lifted[20]. This is due to the necessity to dispatch differently on types from the same parametric family in the absence of the above restriction. Multi-method dispatch itself is quite complicated, and complicating it even further does not seem to provide any additional modeling power. For example, the above situation can be resolved in the following manner:

```
type T_SpecialPersonList subtype of T_List(T_Person) {
    isEmpty() : T_Boolean implementation <function1>;
};
type T_SpecialStringList subtype of T_List(T_String) {
    isEmpty() : T_Boolean implementation <function2>;
};
```

or by using classes:

```
class C_PersonList implements T_List(T_Person) {
    isEmpty() implementation <function1>;
};
class C_StringList implements T_List(T_Person) {
    isEmpty() implementation <function2>;
};
```

If P is a product type of a behavior argument in a behavior application **behavior(args)**, then only certain associations will be relevant. An association

```
behavior(<product-type>) implementation <function>
```

is called *applicable to an argument product type P* iff $P \preceq$ **<product-type>**. Of the two associations 1 and 2 applicable to P, 1 is called *more specific than 2 for an argument type P* iff the product type specified in the association 1 is a subtype of the product type of the behavior association 2.

A behavior is always dispatched according to the *most specific* association for a given argument product type P. Therefore, the type system has to verify that for every type-correct behavior application **behavior(args)**, for every P such that $P \preceq$ (static type of **args**) and all types in P are class types, the set of the most specific applicable associations has exactly one element.

If the set is empty, it means that there are no functions that the system should execute during behavior application - the situation usually referred to as a "message not understood" error. On the other hand, several most specific elements mean that the system can not choose which function to execute — "message ambiguous" error.

The requirement that all types in P must be class types is due to the fact that objects can only exist inside classes. Therefore, any set of arguments that appears during dynamic execution of the program has a type P such that all types in P are class types.

The following example illustrates the discussion above.

[20]The type system of $ML_{\leq}$ [BM96a] which is in many ways similar to the one presented here has a similar restriction.

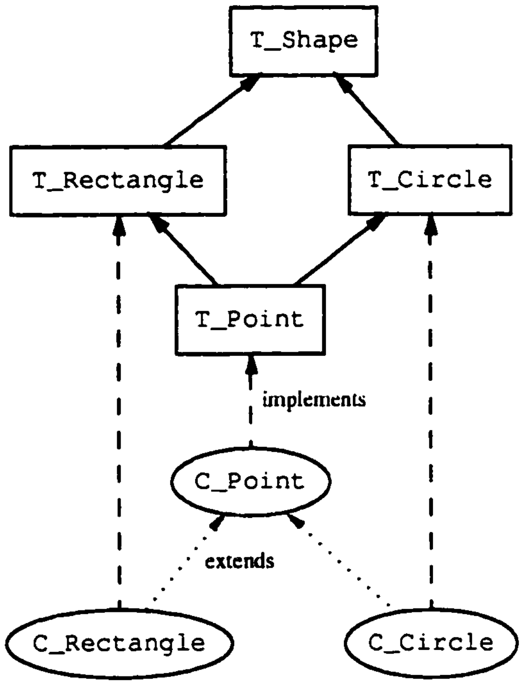

Figure 3.6: Shape type hierarchy

```
type T_Shape {
    center :=: T_Point;
    intersects(T_Shape) : T_Boolean;
};
type T_Rectangle subtype of T_Shape {
    height :=: T_Real;
    width  :=: T_Real;
    intersects(T_Rectangle) : T_Boolean implementation <functionRR>;  // RR
    intersects(T_Circle) : T_Boolean implementation <functionRC>;     // RC
};
class C_Rectangle extends C_Point implements T_Rectangle { ... };
type T_Circle subtype of T_Shape {
    radius :=: T_Real;
    intersects(T_Circle) : T_Boolean implementation <functionCC>;     // CC
    intersects(T_Rectangle) : T_Boolean implementation <functionCR>;  // CR
};
class C_Circle extends C_Point implements T_Circle { ... };
type T_Point subtype of T_Circle, T_Rectangle {
    intersects(T_Point) : T_Boolean implementation <functionPP>;      // PP
};
class C_Point implements T_Point { ... };
```

The specification defines a hierarchy of shape types and implements a set of intersection functions. The type **T_Shape** is an abstract type (there is no class implementing shapes). Types **T_Rectangle**, **T_Circle**, and **T_Point** are concrete types. The type structure for this example is depicted in Figure 3.6.

Consider the following behavior application: **intersects(aPoint, aPoint)**. The set of applicable associations include the associations **RR**, **CC**, and **PP**, of which the third one is the most specific:

$$\text{arguments} = \langle \texttt{C\_Point}, \texttt{C\_Point} \rangle \preceq \langle \texttt{T\_Rectangle}, \texttt{T\_Rectangle} \rangle = \texttt{RR}$$
$$\text{arguments} = \langle \texttt{C\_Point}, \texttt{C\_Point} \rangle \preceq \langle \texttt{T\_Circle}, \texttt{T\_Circle} \rangle = \texttt{CC}$$
$$\text{arguments} = \langle \texttt{C\_Point}, \texttt{C\_Point} \rangle \preceq \langle \texttt{T\_Point}, \texttt{T\_Point} \rangle = \texttt{PP}$$
$$\texttt{PP} = \langle \texttt{T\_Point}, \texttt{T\_Point} \rangle \preceq \langle \texttt{T\_Rectangle}, \texttt{T\_Rectangle} \rangle = \texttt{RR}$$
$$\texttt{PP} = \langle \texttt{T\_Point}, \texttt{T\_Point} \rangle \preceq \langle \texttt{T\_Circle}, \texttt{T\_Circle} \rangle = \texttt{CC}$$

Therefore, the function **<functionPP>** will be executed. If the association **PP** was removed, the set of applicable associations would consist of **RR** and **CC**, none of which is more specific than the other, and the behavior application would be ambiguous.

Assume that the association **CR** is removed. Then, the application **intersects(aCircle, aRectangle)** would not have any applicable associations even though the application is statically type correct.

Note, however, that the absence of applicable associations for the argument type ⟨**T_Shape**, **T_Shape**⟩ does not cause problems since the type **T_Shape** is *abstract* and therefore the argument typed ⟨**T_Shape**, **T_Shape**⟩ can never be an actual argument type for a behavior application.

The algorithms for verifying the restrictions described above will be introduced in Chapter 4, and their formal treatment is in Chapter 5.

In this section, the dispatch process and the requirements related to its consistency have been discussed. Next, higher-order functional constructs and their typing will be considered.

3.2.6 Closures and typeif

Behavior application is a powerful construct that allows a programmer to conveniently express most of the necessary computations. Sometimes, however, a function needs to be introduced for a single computation only. In such cases, creation of a behavior and binding it to the necessary function is too expensive to be tolerated. Another, more concise construct called *a closure* provides the capability to construct such one-use-only functions on the fly.

Consider the following specification of a behavior **map** that applies the given argument behavior to all elements of the receiver set and produces a set of answers:

```
type T_Set(X) {
    map((X):Y) : T_Set(Y);
};
```

Assume the programmer wants to use this behavior to produce a set that consists of all elements of the receiver set of numbers increased by 5. The following code is supposed to perform the task:

```
T_Set(T_Number) aSourceSet, aDestSet;
...
aDestSet := aSourceSet.map(???);
```

but which behavior should be put in the place of ???? The behavior should take a number and produce that number increased by 5. While it is possible to define a new behavior for that purpose:

```
type T_Number {
    add5() : T_Number implementation { return add(5); };
};
...
aDestSet := aSourceSet.map(add5);
```

such a definition is unnecessary since it is only used once.

With closures, the above example can be coded as

```
aDestSet := aSourceSet.map(fun(x) { return x.add(5); });
```

where the anonymous function was introduced on-the-fly. This style of programming is common in functional languages; however, it is rarely used in statically typed object-oriented programming languages. The presented type system not only allows the above example to be programmed, it also *infers* the type of the function from the code of the function. In the example above, the type inferred will be[21] **T_Number →T_Number** provided that the type of the behavior **add** is **T_Number →T_Number**.

Closures allow simplified specification of functions that work the same way on objects of all types they are applicable to. What happens if a behavior that is used only once has to behave differently on different types of arguments? In this case, a special construct **typeif** can be utilized.

Consider the following situation: there is a set of persons, and the task is to print out the names of the persons in the set. However, if a person from the set happens to be a student, the student id number has to be printed as well.

It is possible to define a new behavior **extendedName** and implement it differently for persons and students:

```
type T_Set(X) {
     foreach((X):T_Unit);
};
type T_Person {
    name :=: T_String;
    extendedName() : T_String implementation {
           return name;
        };
};
type T_Student subtype of T_Person {
    studentId :=: T_Natural;
    extendedName : T_String implementation {
           return name + " student ID#" + studentId.convertToString;
        };
};
...
T_Set(T_Person) aSet;
...
let f = fun(x) { x.extendedName.print } in aSet.foreach(f);
```

However, this approach suffers from the same drawback as the first approach used to implement the previous example. Namely, the introduction of a new behavior that is only going to be used once is unwarranted.

Using the **typeif** construct, the above example can be rewritten as

```
T_Set(T_Person) aSet;
...
let f = fun(x) {
                x.name.print;
                typeif (x is T_Student) then {
                   " studentID#".print;
                   x.studentId.print;
                };
               }
in aSet.foreach(f);
```

The **typeif** construct checks if the run-time type of the given expression is a *subtype* of the given type and executes the code given if it is. The code is typechecked with the assumption that the expression is of the type given. This special behavior of the **typeif** construct with respect to type checking is crucial for static type safety. The example above would not typecheck if the typechecker

[21] After certain simplifications.

did not take the given type T_Student into account, as otherwise the type of x would be considered to be T_Person and the behavior application x.studentId would be considered illegal.

The same construct can be used to code the *(BROWSER)* test from Section 2.6:

```
T_Object root;
...
typeif (root is T_Number) then {
    root.print;
} elseif (root is T_Person) then {
    root.age.print;
} else {
    "Something else".print;
};
```

Note the usage of **else** and **elseif** in the above example.

In this section, closures and the **typeif** constructs have been considered. Their use was illustrated by several examples, and the issues of their typing were discussed.

This concludes the overview of the intermediate abstraction layer consisting of types and functions. In the next section, the lowest specification layer will be described.

3.3 Implementation types

Implementation types and functions represent the lowest layer of the presented type system. They are designed to provide support for interoperability and low-level optimizations.

Implementation functions represent what is sometimes referred to as *foreign* or *primitive* functions, i.e. functions that are specified outside of the system. *Implementation types*, on the other hand, serve dual purpose: they provide a low-level structure specification and they allow for a limited form of typechecking of the foreign function interfaces.

The way implementation types and functions are described is implementation-dependent. For example, if the target language is C++, implementation functions are C++ functions, while implementation types are C++ classes. If a target language provides a notion of subtyping, implementation types may have a subtyping relationship between each other; otherwise, they are unrelated.

Implementation types can be used to implement the structure defined by classes. For example, the following is the specification of integers and reals under the assumption that the target language is C++:

```
type T_Real {
    add(T_Real) : T_Real;
};
infinite class C_Real implements T_Real {
    implementation type short atomic float;
    add(C_Real) : C_Real implementation function
        float addRR(float self, float arg) { return self + arg; };
    add(C_Integer) : C_Real implementation function
        float addRI(float self, int arg) { return self + arg; };
};

type T_Integer {
    add(T_Integer) : T_Integer;
};
infinite class C_Integer implements T_Integer {
    implementation type short atomic int;
    add(C_Integer) : C_Integer implementation function
        int addII(int self, int arg) { return self + arg; };
    add(C_Real) : C_Real implementation function
```

```
        float addIR(int self, float arg) { return self + arg; };
};
```

Note that the specification above[22] requires classes rather than types in argument specifications. The reason for that is the fact that the system has to know how to convert objects into low-level data, and that information is only provided by classes.

The system verifies that all possible behavior applications can be dispatched to an appropriate function or implementation function. In the above specification, all four combinations had to be explicitly specified since C++ can not dispatch on the data of type `int`. The system also makes sure that substitutability for subclasses is not violated in case the subclass uses an implementation type that is different from the one used in the superclass.

The usage of implementation types allows seamless integration of foreign data into the existing system. Once an implementation type and a class that describe the foreign objects are specified, those objects can be manipulated by behaviors and functions in exactly the same manner as the objects native to the system. All code previously written will work on foreign data with no modifications.

Another use of implementation types and functions is related to the specification of so-called *primitive types*. Unlike other systems, the system described here is *extensible* in that new primitive types can be easily introduced. The above example can be seen as a definition of integer and real primitive types that is done by a programmer rather than by the programming language designer.

The interoperability and extensibility provided by this system are extremely important for database applications that tend to evolve over long periods of time and incorporate data from different sources, some of which are beyond the database designer's control. Other arguments in favor of implementation types and functions, as well as in favor of the three-layer type system structure in general, can be found in [LÖS98].

An ordinary user may never see or use implementation types and functions. This low-level mechanism is provided for administrative purposes and is not supposed to be used by the ordinary users.

While implementation types and functions are important for interoperability, their role in typechecking, which is the major topic of the presented research, is marginal. Implementation types and functions described here will not be considered in any detail in the following chapters that are devoted primarily to typechecking.

The next section describes the advantages of the three-layer design and its possible applications in a database system.

3.4 Three-layer design: Advantages and applications

The three-layer design described in this chapter not only allows for a high degree of code reuse, but also makes the system more transparent for the user by making it possible to use the system at different abstraction layers, depending on the user's needs and level of expertise.

An ordinary user posing queries against the database using a query language only needs to know about the highest abstraction layer — that of behaviors and types. A query may contain behavior applications (a.k.a. path expressions), and the knowledge of the types and behaviors defined in the system is sufficient for the purpose of constructing such behavior applications. Thus, an ordinary user is completely shielded from any implementation details, such as the layout of the objects involved in the query and the way behaviors are implemented. This has two major advantages: first, the user is presented with a significantly simplified, yet consistent view of the object model. Second, the user is provided with a uniform view of objects and behaviors that is independent of their implementation. For example, if a point type is implemented via two different classes (polar and rectangular points), the path expression

```
    aPoint.x
```

will produce the x-coordinate for both polar and rectangular points, even though the code that is executed in both cases is completely different.

[22]The meaning of the specifier **short atomic** is explained in Section B.1.

An application programmer should be given access to the second abstraction layer consisting of functions and classes. This arrangement would make it possible to implement a new functionality by writing function code and new data structures by defining new classes. Thus, an application programmer is given the opportunity to develop the system. However, an application programmer is shielded from the differences in the operating system functionality and bit-level data structures that may exist between different systems, as this functionality is provided by implementation types and functions that are not visible at the second abstraction layer. The code developed at this abstraction layer is therefore system-independent and portable. In addition to this, if a piece of code in an existing system is modified (e.g., a more efficient implementation of a particular behavior is produced), such modification does not affect the validity of the user code written at the first abstraction layer, as the interfaces are not affected.

Finally, a database designer would have access to the third abstraction layer (implementation types and functions). Using this layer involves dealing with native code, which makes it possible to handle tasks involving low-level system access, access to foreign data, and introduction of new optimized primitive types (e.g., multimedia types such as audio and video). It should be noted that any development done at this layer does not affect the validity of the code written earlier by either ordinary users or application programmers.

3.5 The basic type system

This section describes an application of the principles and constructs described in this chapter. The basic type system for the TIGUKAT object model [Pet94] has been developed as a proof of concept and the basis for the development of the TIGUKAT programming language.

Only object types, atomic types, and container types will be described here. The complete specification of the basic TIGUKAT type system including reflexive metaclass and metatype specifications can be found in Appendix A.

3.5.1 Object types

In the presented type system, all types can be separated into three basic categories: *object types*, *function types*, and *product types*. The types from different categories do not have any subtyping relationships with each other. This structure is depicted in Figure 3.7. The types $\top$ and $\perp$ are purely theoretical concepts; they can not be used by the user in any way. In fact, the type $\top$ can be understood as a type of all type errors that are guaranteed not to occur in a successfully typechecked program.

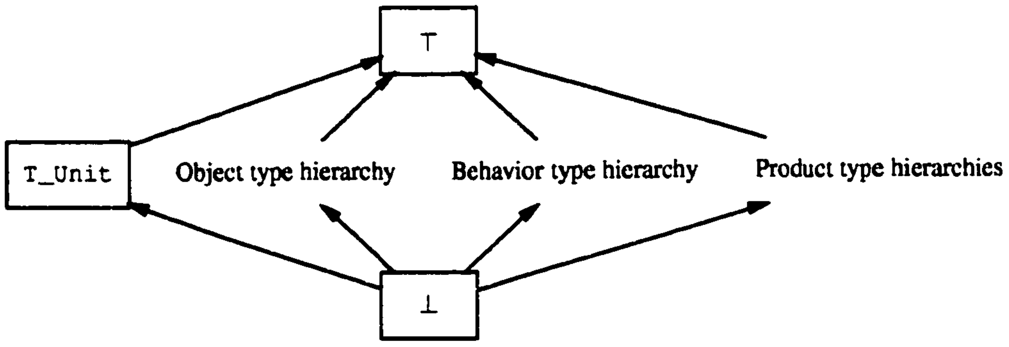

Figure 3.7: TIGUKAT types.

Object types are the types of all objects in the system except for functions (behaviors) and products. This is the type category that includes almost all user- and system-defined types in the system. This category will be discussed in detail later.

Product types are types of *products* that are ordered tuples of values. Subtyping between product types is induced by the subtyping between their component types. n-ary product types are defined for all n greater than 1. Product types of different arity do not have any subtyping relationship with each other. Thus, there are several unrelated product type hierarchies: one for each $n > 1$. A special type T_Unit plays the role of a 0-ary product: it has no subtypes or supertypes. It is used primarily for specifying return types of behaviors that are not supposed to return a result. n-ary products are assumed to have the following definitions:

```
type T_ProductN(covar X1, covar X2, ..., covar XN) {
    project1(): X1;
    project2(): X2;
    ...
    projectN(): XN;
};
```

for each N greater than 1.

Function and behavior types are defined as follows:

```
type T_Function(contravar A, covar R);
type T_Behavior(contravar A, covar R) subtype of T_Function(A,R);
```

These are the only functional types in the system.

In the object type hierarchy, there are two types that play a special role. They are T_Object and T_Null.

The type T_Object is a common supertype of all object types. It does not define any behaviors, but they can be added to T_Object by the user.

The type T_Null is a common *subtype* of all object types. Therefore, values of type T_Null can be used everywhere in the program. The values of type T_Null denote uninitialized, missing, or otherwise absent (null) values. Thus, T_Null is a concrete type.

Figure 3.8 depicts the relative placement of various object types in the object type hierarchy.

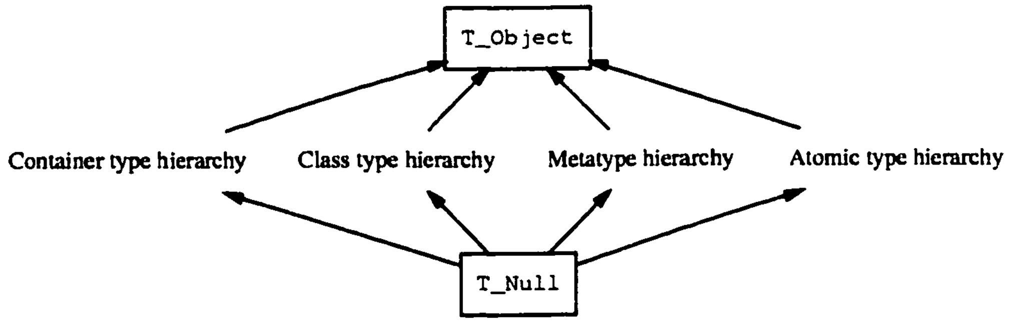

Figure 3.8: TIGUKAT object types.

Even though there is no common supertype of *all* types in the system (T_Object is a common supertype of *object types*, but not *all types*), the parametric nature of the type system allows for specification of behaviors that are uniformly applicable to *all* objects, including products and behaviors. The following is the specification of these behaviors:

```
behavior OIDequal(X,X) : T_Boolean;     // Object identity equality
behavior equal(X,X)    : T_Boolean      // Equality
                                        // Shortcut: operator binary "=="
        implementation fun(x,y) { return x.OIDequal(y); };
notEqual(X,X)          : T_Boolean      // Inequality
                                        // Shortcut: operator binary "!="
        implementation fun(x,y) { return not (x == y); };
print(X,OutputStream(X));               // Print to an output stream
apply((X):Y) : Y;                       // Behavior application
                                        // Shortcut: operator binary "."
```

Due to the fact that in the presented system the notations `f(a,b)` and `a.f(b)` are equivalent and interchangeable, the above code for equality and inequality is type-correct and behaves as expected.

The following is the specification of the two special object types.

```
type T_Object;
type T_Null subtype of <all object types>;
class C_Null implements T_Null { ... };
```

3.5.2 Atomic types

Atomic types include the standard set of real, integer, and natural numbers, characters, and booleans. Abstract types **T_Numeric**, **T_PartiallyOrdered**, **T_Ordered**, and **T_Discrete** abstract out several common properties that are likely to be reused for future extensions.

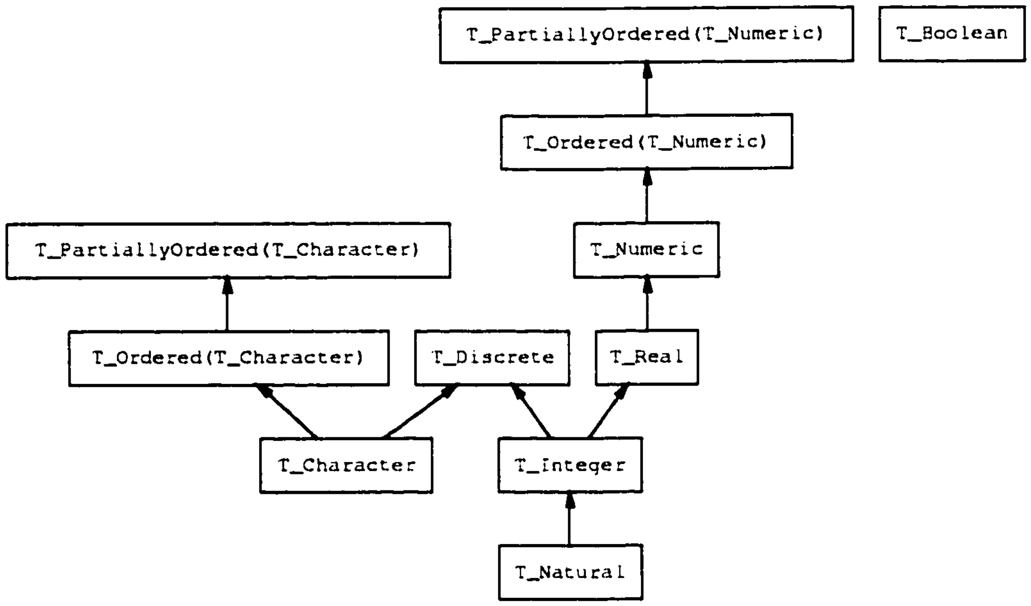

Figure 3.9: TIGUKAT atomic types.

```
type T_Discrete subtype of T_Object {
    pred : selftype;                // Return the previous element
    succ : selftype;                // Return the next element
};

type T_PartiallyOrdered(contravar X) subtype of T_Object {
    less(selftype) : T_Boolean;            // Antisymmetric comparison.
                                           // Shortcut: operator binary '<'
    greater(selftype) : T_Boolean          // Antisymmetric comparison.
                                           // Shortcut: operator binary '>'
        implementation fun(x) { return x < self; };
    lessOrEqual(selftype) : T_Boolean      // Symmetric comparison.
                                           // Shortcut: operator binary '<='
        implementation fun(x) { return x < self or x == self; };
    greaterOrEqual(selftype) : T_Boolean   // Symmetric comparison.
                                           // Shortcut: operator binary '>='
        implementation fun(x) { return x <= self; };
};
```

```
type T_Ordered(contravar X) subtype of T_PartiallyOrdered(X) {
   max(selftype) : selftype             // Maximum of the two
       implementation fun(x) {
          return if x < self then self; else x; endif };
   min(selftype) : selftype             // Minimum of the two
       implementation fun(x) {
          return if x < self then x; else self; endif };
};

type T_Numeric subtype of T_Ordered(T_Numeric) {
   abs : T_Numeric;                   // Absolute value
   negate : T_Numeric;                // Negation.
                                      // Shortcut: operator unary '-'
   add(T_Numeric) : T_Numeric;        // Addition.
                                      // Shortcut: operator binary '+'
   subtract(T_Numeric) : T_Numeric;   // Subtraction.
                                      // Shortcut: operator binary '-'
   multiply(T_Numeric) : T_Numeric;   // Multiplication.
                                      // Shortcut: operator binary '*'
   divide(T_Numeric) : T_Numeric;     // Division.
                                      // Shortcut: operator binary '/'
};
```

So far, only abstract types have been specified. Note how F-bounded quantification ([CCH+89], Section 3.1.7) is used in ordered type specifications to disallow unwanted cross-comparisons, such as comparisons between characters and numbers. The specification of ordered types also provides a default implementation for all behaviors except for `less`. Once the implementation of that behavior is provided, all the other comparison behaviors are automatically available.

```
type T_Boolean {
   not : T_Boolean;                   // Negation.
   or(T_Boolean) : T_Boolean;         // Logical OR.
                                      // Shortcut: operator binary 'or'
   and(T_Boolean) : T_Boolean;        // Logical AND.
                                      // Shortcut: operator binary 'and'
   xor(T_Boolean) : T_Boolean;        // Logical XOR.
                                      // Shortcut: operator binary 'xor'
   if(X, X)       : X;                // If-expression. Shortcut:
                                      // 'if ... then ... else ... endif'
};

finite class C_Boolean implements T_Boolean {
   implementation type short atomic int;
   not implementation function
       not(int self) { return ~self; };
   or(C_Boolean) implementation function
       and(int self, int x) { return self || x; };
   ...
};

type T_Character subtype of T_Discrete, T_Ordered(T_Character) {
   ord  : T_Natural;                  // Returns the ordinal value
};
```

```
finite class C_ASCIICharacter implements T_Character {
    implementation type short atomic char;
    ...
};

finite class C_UnicodeCharacter implements T_Character {
    implementation type short atomic Unichar;
    ...
};
```

The above types are concrete. All of the described classes are **finite** since they do not allow object creation, yet maintain their (finite) extents. The parameterized behavior **if** defined on booleans is typed in such a way as to provide the most static type information possible (it is equivalent to **if(X, Y) : lub(X, Y)**). There are two separate classes implementing characters: a standard ASCII character and a Unicode character. Since these are *classes* rather than *types*, any program written will be able to work seamlessly with both types of characters or any mix of them.

```
type T_Real subtype of T_Numeric {
    truncate : T_Integer;           // Truncate throwing away
                                    // the fractional part
    round    : T_Integer;           // Round to the nearest
    sign     : T_Integer;           // Sign: 1 if positive, -1 if
                                    // negative, 0 otherwise
    // Refined from T_Numeric
    abs : T_Real;
    negate : T_Real;
    add(T_Real) : T_Real;
    subtract(T_Real) : T_Real;
    multiply(T_Real) : T_Real;
    divide(T_Real) : T_Real;
};

infinite class C_Real implements T_Real {
    implementation type short atomic float;
    ...
};

infinite class C_LongReal implements T_Real {
    implementation type long atomic double;
    ...
};

type T_Integer subtype of T_Real, T_Discrete {
    div(T_Integer) : T_Integer;          // Integer division

    // Refined from T_Real
    abs : T_Natural;
    negate : T_Integer;
    add(T_Integer) : T_Integer;
    subtract(T_Integer) : T_Integer;
    multiply(T_Integer) : T_Integer;
};

infinite class C_Integer implements T_Integer {
    implementation type short atomic int;
    ...
};

infinite class C_LongInteger implements T_Integer {
    implementation type long atomic long int;
    ...
};

type T_Natural subtype of T_Integer {
    mod(T_Natural) : T_Natural;          // Remainder

    // Refined from T_Integer
    truncate : T_Natural;
    round : T_Natural;
    sign : T_Natural;
    div(T_Natural) : T_Natural;
    add(T_Natural) : T_Natural;
    multiply(T_Natural) : T_Natural;
};

infinite class C_Natural implements T_Natural {
    implementation type short atomic unsigned;
    ...
};

infinite class C_LongNatural implements T_Natural {
    implementation type long atomic long unsigned;
    ...
};
```

The above is the specification of numeric types. Each of the types is implemented by several classes; all classes are **infinite** as they have infinite extents and do not allow object creation. Careful redefinition of binary methods along the type hierarchy allows for a very precise typing. For example, the system will deduce that 5 + 6 has the type T_Natural, while 5 + 6.0 has the type T_Real.

Any two numerics can be compared using comparison operations, including possible **min** and **max** operations, since all numeric types are subtypes of T_Numeric and therefore T_Ordered(T_Numeric). Integers are also discrete (subtype of T_Discrete), and therefore the behaviors **pred** and **succ** can be applied to them.

3.5.3 Collection types

The collection type hierarchy is represented here only partially. Its full description can be found in Appendix A. The hierarchy described here is depicted in Figure 3.10.

```
type T_Collection(covar X) {
    hasElement(X) : T_Boolean;        // Checks if the element
                                      // is in the collection
    cardinality   : T_Natural;        // Cardinality of the collection
    pick          : X;                // Pick an element
    map((X):Y)    : T_Collection(Y);  // Apply the given function to all
                                      // elements and return the
                                      // collection of results
    filter((X):T_Boolean) : T_Collection(X); // Filters the receiver
                                      // collection using the
                                      // argument function as a filter
```

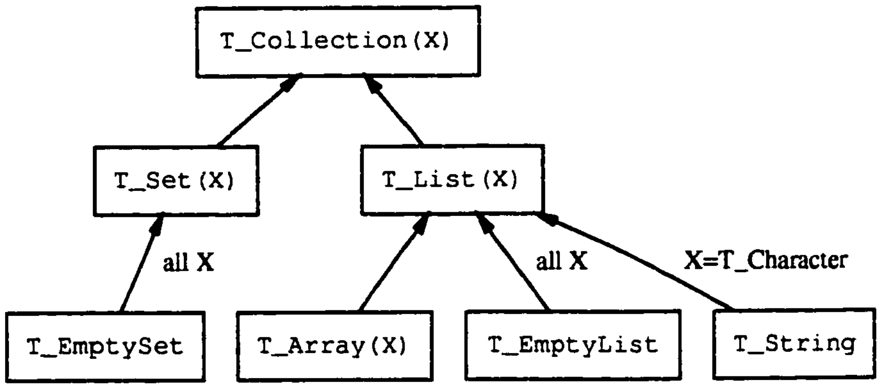

Figure 3.10: TIGUKAT collection types.

```
    sort() where (X subtype of T_Ordered(X)): T_List(X); // Sorts the
                                              // elements of the collection.
                                              // Only applicable to
                                              // collections of ordered elements
    sort((X,X) : T_Boolean) : T_List(X); // Sorts the collection using
                                              // the supplied sorting function
};

type T_Set(covar X) subtype of T_Collection(X) {
    subset(T_Set(Y))     : T_Boolean;        // Is the receiver
                                              //  a subset of the argument?
    union(T_Set(Y))      : T_Set(lub(X,Y)); // Set union
    intersect(T_Set(Y)) : T_Set(glb(X,Y)); // Set intersection
    difference(T_Set(Y)): T_Set(X);         // Set difference
    addElement(Y)        : T_Set(lub(X,Y)); // Add an element
    removeElement(Y)     : T_Set(X);         // Remove an element
    // Refined from T_Collection
    map((X):Y)    : T_Set(Y);
    filter((X):T_Boolean) : T_Set(X);
};

infinite class C_Set(X) implements T_Set(X) {
    const T_List(X) _list;
    ...
};

type T_EmptySet subtype of T_Set(X);

finite class C_EmptySet implements T_EmptySet { ... };

type T_List(covar X) subtype of T_Collection(X) {
    at(T_Natural)       : X;                   // Get at arg's position
    cat(T_List(Y))      : T_List(lub(X, Y)); // Concatenation
    slice(T_Natural, T_Natural) : T_List(X); // Slice from i-th
                                              // to j-th element
    // Refined from T_Collection
    map((X):Y)    : T_List(Y);
    filter((X):T_Boolean) : T_List(X);
```

```
};

infinite class C_List(X) implements T_List(X) { ... };

type T_EmptyList subtype of T_List(X);

finite class C_EmptyList implements T_EmptyList { ... };

type T_Array(novar X) subtype of T_List(X) {
    at(T_Natural)         := X;                  // Set at arg's position
    slice(T_Natural, T_Natural) := T_List(X); // Slice set
};

manual class C_Array(X) implements T_Array(X) { ... };

type T_String subtype of T_List(T_Character) {
    // Refined from T_List
    cat(T_String)        : T_String;
    slice(T_Natural, T_Natural) : T_String;
    filter((T_Character):T_Boolean) : T_String;
};

infinite class C_String implements T_String { ... };

infinite class C_UnicodeString implements T_String { ... };
```

This specification defines several immutable parameterized collection types and a mutable type **T_Array**. The type of empty sets (lists) is a subtype of all set (list) types, so that a single empty set (list) can be used in all set (list) operations. The type **T_String** is defined as a subtype of the character list type that has some additional properties.

The set-theoretic and higher-order (map, filter, sort) behaviors defined on collections have precise typings. This property can be used to define a statically typechecked version of a query language over immutable sets.

The slice assignment defined on arrays allows one to write the following code:

```
T_Array(T_Number) arr;
arr.slice(3,5) := (1,2,3); // Assign to elements 3,4, and 5
```

Note that the class that implements arrays is specified as **manual**. It allows creation of new arrays, but does not maintain the set of all arrays created so far, as that would be very expensive.

Note also that the class of sets uses **T_List** for implementation of sets. The ability to specify relationships like this is a consequence of the separation between interface and implementation in the presented type system.

This concludes the overview of the basic TIGUKAT type system that was presented as an example of the application of the presented type system to an object-oriented object model. The resulting basic type system provides precise typing for many traditionally problematic areas of object-oriented specification: covariant arguments, query typing, parametricity, and binary methods. Yet, the resulting system is statically type-safe and has the substitutability property.

3.6 Conclusions

In this chapter, an overview of the proposed type system has been presented. Three layers of specification (types and behaviors, classes and functions, implementation types and functions) have been considered. Their use was exemplified with a number of examples. In the process, it has been shown that the presented type system can successfully deal with all the tests described in Section 2.6. Finally, an example application of the developed principles to the TIGUKAT object model has been described.

In the next chapter, the typechecking algorithm will be described. The theorems and proofs of its correctness and termination will be considered in Chapter 5.

Chapter 4

The Language and Typechecking

One of the most important constituent parts of any type system is its *typechecking algorithm*. The typechecking algorithm allows a translator to verify that the program conforms to the given specifications and issue an error message if it does not. A *verifiable* type system has a typechecking algorithm that is guaranteed to terminate even if a program being checked contains type errors. On the other hand, a *sound* type system has the property that once a program successfully passes through a typechecker, it is guaranteed not to produce any type errors at run-time.

The type system presented here is both verifiable and sound. These properties will be proven in Chapter 5 on the basis of the typechecking algorithm presented in this chapter.

This chapter serves to describe two important parts of the type system: the typechecking algorithm (Section 4.2) that works on a simplified version of the language (*the target language* – Section 4.1.1), and the "de-sugaring" translation process (Section 4.1.2) that produces a program in the target language from a given source language program.

The heart of the typechecking algorithm is the algorithm that decides entailment (Section 4.3). This algorithm is language-independent and is part of the type system core.

In the next section, both source and target languages will be considered and the translation process will be described.

4.1 Syntax and translation

Since the beginning of the computer era, there has been a gap between an "efficient" and "human-readable" presentation of information. The first form of information presentation is easily processed by the computers; on the other hand, the second form is advantageous to human beings. The use of computers to automate the transformation between the two forms of information representation has been a cornerstone of the computer revolution.

In this section, the *source* and *target* (Section 4.1.1) languages will be described. The source language has a traditional syntax which is familiar to people who deal with procedural and object-oriented programming. The target language's syntax is less standard; however, it is much more concise and significantly more convenient to use in automatic typechecking.

Note that this chapter does *not* present a complete source language design. Rather, it is a proof-of-concept design that shows how the presented type system can be used to model various features existing in today's programming languages.

Finally, in Section 4.1.2 the process of automatic translation from source to target language will be described. The translation process is sometimes called *de-sugaring* since it removes "syntactic sugar" from the program, making it more suitable for further automatic processing.

4.1.1 The target language

The target language is a "de-sugared", simplified object-oriented language that will be used for typechecking. It consists of type and subtype definitions, behavior declarations, and function associations. Class specifications are translated to type definitions by the process described in Section 4.1.2.

A program in the target language consists of type definitions, subtype definitions, behavior definitions, associations, and an expression:

```
<program> ::= [ <defs> [ ";" ] ] <expr>
<defs> ::= <def> {";" <def>}*
<def> ::= <type-def> | <subtype-def> | <constant-def>
          | <behavior-def> | <association>
```

Type definitions describe types, variance of their parameters, and validity conditions. They also specify whether a type is *concrete* or *abstract*:

```
<type-def> ::= [ <concrete-spec> ] "type" [ "(" <constraints> ")" ]
                     <name> [ "(" <type-par-specs> ")" ] ";"
<concrete-spec> ::= "concrete"
<type-par-specs> ::= <type-par-spec> {"," <type-par-spec>}*
<type-par-spec> ::= <variance> <name>
<variance> ::= "covar" | "contravar" | "novar"
<constraints> ::= <constraint> {"," <constraint>}*
<constraint> ::= <type> "<=" <type>
```

The following are examples of type specifications:

```
type T_Number;
concrete type T_Behavior(contravar Arg, covar Result);
concrete type T_Product2(covar X1, covar X2);
concrete type T_Set (covar X);
concrete type (X <= T_Ordered(X)) T_OrderedSet (covar X);
```

A type can be defined only once. It is illegal to have two definitions of the same type (type with the same name).

Types that are used in specifications are defined as follows:

```
<type> ::= <name> [ "(" <types> ")" ]
         | "glb" "(" <types> ")"
         | "lub" "(" <types> ")"
<types> ::= <type> {"," <type>}*
<type-proper> ::= <name> [ "(" <names> ")" ]
<names> ::= <name> {"," <name>}*
```

Examples of types are

```
T_Number
T_Behavior(T_Product2(T_Set(X), T_Set(Y)), T_Set(lub(X,Y)))
T_Dictionary(X,T_Dictionary(T_Number,X))
```

The shortcut `<typeA> -> <typeR>` will be used for `T_Function(<typeA>, <typeR>)` (functional types), and the shortcut `(<type1>,...,<typeN>)` will be used for `T_ProductN(<type1>,...,<typeN>)` (product types). It will also be assumed that `(<type>)` ≡ `<type>` (unary products and types are indistinguishable).

In the following presentation, the concept of *free type variables* will be extensively used. The set of free type variables in a type expression is defined as a set of `<name>`s in that expression minus the set of all `<name>`s defined in type definitions and is denoted as $FTV(\texttt{t})$. For example, the set of free type variables of the expression `(T_Set(X), T_Set(Y)) -> T_Set(lub(X,Y))` is $\{\texttt{X},\texttt{Y}\}$. A type expression that has an empty set of free type variables will be called a *closed type* (e.g. `T_Number`).

Subtype definitions define subtyping relationships between types. A subtype definition has the form

```
<subtype-def> ::=
     [ "(" <constraints> ")" ] <simple-type> "<=" <simple-type>
<simple-type> ::= <name> [ "(" <types> ")" ]
<simple-types> ::= <simple-type> {"," <simple-type>}*
```

The requirement that only simple types can occur in subtype definitions is designed to protect against user-defined subtyping of upper and lower bound types. It is also assumed that no subtype definitions involve `T_Function`, `T_Behavior`, and `T_Product` at the outermost level, i.e. that the user is not allowed to define any additional subtyping relationships for these types. Syntactically valid examples of subtype definitions are

```
T_Real <= T_Number;
(X <= T_Ordered(X)) T_Set(X) <= T_Ordered(T_Set(X));
```

Constant definitions are used to declare constants and their types.

```
<constant-def> ::= "const" <type> <name> ";"
```

Examples of constant definitions are

```
const T_Class(T_Person) C_Person;
const T_Integer nargs;
```

A constant can be defined only once. It is illegal to have two definitions of the same constant. The `<type>` of a constant should be a closed one.

Behavior definitions are used to declare behaviors to be used in the program. They specify argument and result types of a behavior as well as validity conditions:

```
<behavior-def> ::= <behavior-def-proper> ";"
<behavior-def-proper> ::=
    "behavior" [ "(" <constraints> ")" ] <type> "->" <type> <name>
```

The type on the left-hand side of -> should not contain type operators `lub` and `glb`. Examples of behavior definitions are

```
behavior T_Number -> T_Number add;
behavior (X <= T_PartiallyOrdered(X)) T_Set(X) -> T_Set(X) maximals;
```

There can be several definitions of the same behavior. For example, the following definitions can appear in the same program:

```
behavior (T_Number, T_Number) -> T_Number add;
behavior (T_Real, T_Real) -> T_Real add;
behavior (T_Integer, T_Integer) -> T_Integer add;
behavior (T_Natural, T_Natural) -> T_Natural add;
```

This last property is the reason why behavior and constant definitions are separated from each other. While duplicate constant definitions are disallowed, duplicate behavior definitions are possible, and the system is responsible for checking their consistency and finding out the most specific type for each behavior.

Associations are used to associate an expression to a behavior on a set of types. The associations thus provide a link between a behavior specification and its implementation.

```
<association> ::= <behavior-def-proper> <abstraction-expr> ";"
                | <behavior-def-proper> "primitive" <name> ";"
```

The difference between associations and behavior definitions is analogous to the difference between abstract and concrete types. However, the former is much more relevant to the typechecking than the latter. The *primitive* associations are provided as a tool that enables the language designer to provide the primitive, low-level function definitions to base the language upon. These functions correspond to implementation functions of the source language. The conditions that the primitives must satisfy for the type system to be sound are outlined in Section 5.3.2.

Expressions constitute the action part of a program. There are several kinds of expressions, described below.

```
<expr> ::=    <name>
            | <product-expr>
            | <projection-expr>
            | <let-expr>
            | <compound-expr>
            | <application-expr>
            | <abstraction-expr>
```

Product expressions are used to group several objects into one. Product expressions are primarily used to bypass the single-argument restriction for behaviors and abstractions.

```
<product-expr> ::= "(" <expr> {"," <expr>}* ")"
                 | "{" <expr> {"," <expr>}* "}"
```

A product expression with a single component is equivalent to that component. For example, (a, b) is a binary product expression, while (a) is a unary product expression equivalent to a.

Projection expressions are used to extract components from product expressions. These expressions are not primitive; rather, they are understood as a shortcut for projection behavior applications.

```
<projection-expr> ::= <expr> "_" <number>
<number> ::= {0|1|2|3|4|5|6|7|8|9}+
```

For example, (a, b)_2 $\equiv$ b, while (a, b)_3 is illegal. An expression <expr>_<number> is treated as project<number>(<expr>), where projection behaviors are defined as follows:

```
behavior project1 (X1,...,Xn) : X1 // For all n > 1
behavior project2 (X1,...,Xn) : X2 // For all n > 2
...
behavior projectN (X1,...,Xn) : XN // For all n > N
...
```

Let expressions make it possible to introduce a new name in the environment. Both typed and untyped variants are allowed; if a variable is untyped, its type is *inferred* from the context.

```
<let-expr> ::= "let" [<type>] <name> "=" <expr> "in" <expr>
```

<type> must be a closed type. An abbreviation with multiple bindings will be used to denote a sequence of nested let expressions. For example,

```
let a = f(x), b = g(x) in a(b)
```

is equivalent to

```
let a = f(x) in let b = g(x) in a(b)
```

Compound expressions are used to compute expressions in sequence, discarding the results of all but the last one.

```
<compound-expr> ::= <expr> {";" <expr>}*
```

For example, the result of `a; b; c` is `c`.

Abstractions are anonymous functions (λ-abstractions). They construct *closures* that can be executed immediately or later on.

```
<abstraction-expr> ::=  "fun" "(" [ <names> ] ")"
                            "{" [ <expr> ] [ "return" <expr> ";" ] "}"
```

For example,

```
fun(x) { return add(x,5); }
```

is an abstraction that creates a closure that will be able to add 5 to its argument and return the result. Note that there is always a single argument, which may or may not be a product. The keyword **return** is used to specify the fact that something is returned from the function. If it is absent, the function is presumed to implicitly return a special object `unit` of type `T_Unit`.

Applications are behavior and abstraction applications to a given argument. They are denoted by juxtaposition and associate to the left:

```
<application-expr> ::= <expr> <expr>
```

For example, `f a b` is equivalent to `(f (a)) (b)`. Note that the last expression makes use of unary products of the form `(x)`. Multi-argument function and behavior applications can be similarly expressed via n-ary product expressions: `add(a, b)`.

In this section, the target language has been described. The target language is designed to be an intermediate language that has a form convenient for type-checking. In the next section, the translations of various features of the source language into the target language will be described.

4.1.2 The translation

The source language has been informally introduced in Chapter 3. This section describes a translation of the source language into the target language described in the previous section.

Type specifications in the source language define types, their subtyping relationships, behaviors applicable to the objects of these types, and behavior-to-function associations. The following is the description of the translation process for the type specifications of the source language, whose syntax is given below:

```
<type-def> ::= "type" <type-def-proper> [ "{"
                      ["where" <constraints>]
                      ["subtype of" <type-spec-list>]
                      [ <behavior-def-list> ] "}" ] ";"
<type-def-proper> ::=
   <type-name> ["(" <type-par-spec> {"," <type-par-spec>}* ")"]
<type-par-spec> ::= {"covar" | "contravar" | "novar"} <name>
```

Explicit receiver: Every behavior specification for the type is transformed into a stand-alone behavior definition by adding the receiver as an explicit first argument. The added argument is given the type being defined if **selftype** is not used anywhere in the behavior specification. Otherwise, **selftype** is replaced by a fresh type variable `S`, and the constraint **S subtype of <type-being-defined>** is added to the **where** clause of the behavior specification. Example:

```
type T_Set(covar X) {
   union(selftype): selftype;
   pick(): X;
};
```

is transformed into

```
type T_Set(covar X);

union(S,S): S where S subtype of T_Set(X);
pick(T_Set(X)) : X;
```

Note how the explicit first argument typing depends on the presence or absence of **selftype**.

Assignment elimination: For all behavior definitions that use the symbol :=:, the behavior definition is split into two: one identical to the original, but with symbol : in place of :=:, and the other with the name obtained from the original one by appending the prefix **set_**, the additional argument of the return type, and the new return type T_Unit. Behavior definitions that use the symbol := get transformed into the second alternative outlined above. For example:

```
age(T_Person) :=: T_Natural;
put(T_OutputStream(X)) := X;
substr(T_String, T_Natural, T_Natural) :=: T_String;
```

gets transformed into

```
age(T_Person) : T_Natural;
set_age(T_Person, T_Natural) : T_Unit;

set_put(T_OutputStream(X), X) : T_Unit;

substr(T_String, T_Natural, T_Natural) : T_String;
set_substr(T_String, T_Natural, T_Natural, T_String) : T_Unit;
```

Subtype separation: Each type description is split into the type specification proper and zero or more subtype specifications. **where**-clauses related to the type being defined stay with the type, while **where**-clauses related to a type in the appropriate **subtype of**-clause are kept with the subtype specification that is produced from the clause. For example:

```
type T_OrderedPersonSet(covar X) where (X subtype of T_Person)
     subtype of T_Set(X)
            where (X subtype of T_Ordered(X), T_SpecialSet(X));
```

gets transformed into

```
type T_OrderedPersonSet(covar X) where X subtype of T_Person;
T_OrderedPersonSet(X)  subtype of T_Set(X)
            where X subtype of T_Ordered(X);
T_OrderedPersonSet(X)  subtype of T_SpecialSet(X);
```

Reflexive constants: Since types are objects (of type T_Type) they are not only used for specification purposes, but can also be queried at run-time. Therefore, for each type specification, the constant declaration

```
const T_Type <name>;
```

is added to the program, where **<name>** is the name of the type being specified. For example: the type specification

```
type T_Number;
```

results in the addition of the declaration

```
const T_Type T_Number;
```

to the program.

Syntactic transformations: Finally, the syntax of the declarations is modified to fit the syntax of the target language. Namely,

1. **where**-clauses are transformed into lists of `<constraints>`;
2. the behavior specifications get transformed from
   `<name>(<arg-types>) : <res-type>;`
   into
   `behavior (<arg-types>) -> <res-type> <name>;`
3. behaviors with **implementation**-clauses get transformed into associations.

For example:

```
type T_OrderedPersonSet(covar X) where X subtype of T_Person;
T_OrderedPersonSet(X)  subtype of T_Set(X)
                where X subtype of T_Ordered(X);
T_OrderedPersonSet(X)  subtype of T_SpecialSet(X);

union(T_OrderedPersonSet(X),
      T_OrderedPersonSet(X)) : T_OrderedPersonSet(X);
add(T_Integer r, T_Integer a) : T_Integer implementation { ... };
```

is transformed into

```
type (X <= T_Person) T_OrderedPersonSet(covar X);
(X <= T_Ordered(X)) T_OrderedPersonSet(X) <= T_Set(X);
T_OrderedPersonSet(X) <= T_SpecialSet(X);

behavior (T_OrderedPersonSet(X),
          T_OrderedPersonSet(X)) -> T_OrderedPersonSet(X) union;
association (T_Integer, T_Integer) -> T_Integer add
          fun(r, a) { ... };
```

This concludes the description of the type definition translation.

Class specifications in the source language define classes, their abstract structure (fields), the class type (infinite, finite, manual, or regular), subclassing and class extension, behaviors defined on classes, and possibly their low-level structure. Syntax of the source language class definition is given below:

```
<class-def> ::= [ <class-specifier> ] "class" <class-proper> [ "{"
                      ["implements" <type-proper>]
                      ["subclass of" <classes>]
                      ["extends" <class-ext-list>]
                      [ <field-def-list> ]
                      [ <behavior-def-list> ] "}" ] ";"
<class-specifier> ::= "infinite" | "finite" | "manual" | "regular"
<class-proper> ::= <class-name>["(" <name> {"," <name>}*")"]
<class-ext-list> ::= <class-ext> {"," <class-ext>}*
<class-ext> ::= <class-spec> [ "removing" <name-list> ]
<class-spec> ::= <class-name> ["(" <class-spec> {"," <class-spec>}*")"]
<class-name> ::= <name> | "selfclass"
<classes> ::= <class-proper> { "," <class-proper> }*
```

Class extension: A class extension is a mechanism for textual substitution. Therefore, cyclic extension dependencies between classes are not allowed. If such a dependency is detected, it is an error and the translation stops. Otherwise, classes form a directed acyclic graph (DAG) with respect to the relationship **extends**. Starting with the roots of the DAG and following the graph edges in the direction opposite to the one given by the **extends** relationship in such a way that a node is visited only when all its parents are visited (width-first traversal), the following operation is performed. A class that **extends** another class is expanded by copying all field and behavior definitions from the class being extended except for the ones whose names are specified in the appropriate **removing** clause, marking the copied definitions in the process. Thus, *inherited* definitions are marked while *native* ones are unmarked. After the traversal is finished, every class that contains marked definitions is examined. If there are identical definitions, extra copies are removed; if there are conflicting definitions and at least one of them is unmarked, the marked definitions are removed; if there are conflicting definitions and none of them is unmarked, it is an error and the translation stops. The definitions are conflicting if they define a behavior or field with the same name. If there are conflicting unmarked definitions for a field, it is also considered an error. Finally, **extends** clauses are removed from all classes.

As an example, consider the following class specifications:

```
class C_Circle implements T_Circle {
    T_Point _center;
    T_Length _majorAxis;
    center implementation _center;
    majorAxis implementation _majorAxis;
    minorAxis implementation _majorAxis;
};
class C_Ellipse extends C_Circle implements T_Ellipse {
    T_Length _minorAxis;
    minorAxis implementation _minorAxis;
};
```

At the first stage, it is determined that there is no cyclic dependency and `C_Circle` is the root of the DAG. The edge from `C_Ellipse` to `C_Circle` is then traversed, and the class `C_Ellipse` is extended. After copying, it will take the form[1]

```
class C_Ellipse extends C_Circle implements T_Ellipse {
    *T_Point _center;
    *T_Length _majorAxis;
    *center implementation _center;
    *majorAxis implementation _majorAxis;
    *minorAxis implementation _majorAxis;
    T_Length _minorAxis;
    minorAxis implementation _minorAxis;
};
```

Then, since there are two distinct definitions of the behavior **minorAxis**, the marked (inherited) one is discarded. Finally, the **extends** clause is removed and the resulting class takes the form

```
class C_Ellipse implements T_Ellipse {
    T_Point _center;
    T_Length _majorAxis;
    center implementation _center;
    majorAxis implementation _majorAxis;
    T_Length _minorAxis;
    minorAxis implementation _minorAxis;
};
```

Note that unmarked (native) definitions take precedence over marked (inherited) definitions. Note also that identical definitions never cause errors, even if there are no native ones: additional copies are simply discarded. This makes it possible to have diamond-shaped extensions; its behavior is similar to that of the mechanism of *virtual classes* in C++ [Str91].

[1]Marked (inherited) definitions are identified by *.

Field elimination: Field definitions in classes are translated to behavior definitions. Before it is done, the check for subclass conformance is performed; if subclass dependencies are cyclic, it is an error. Otherwise, classes form a DAG with respect to the relationship **subclass of** and for every class, there are finitely many superclasses in the graph. Then, for each class and each superclass, it is verified that if a field is defined in a superclass, its definition is either absent from the subclass, or has the same type, or the field is constant in the superclass and its definition in the subclass specifies a type (class) that is a subtype (subclass) of the one specified in the superclass. This rule is based on the observation that constant (non-updatable) fields are covariant, while updatable fields are novariant. For example, the following specification is legal

```
type T_SmallNatural subtype of T_Natural;
class C_Person {
    const T_Natural _age;
};
class C_Child subclass of C_Person {
    T_SmallNatural _age;
};
```

while the one below is not:

```
type T_SmallNatural subtype of T_Natural;
class C_Person {
    T_Natural _age;
};
class C_Child subclass of C_Person {
    T_SmallNatural _age;
};
```

After the check above has been successfully performed, constant field definitions of the form

```
const <type> <name>
```

are translated into

```
<name>() : <type> implementation fun(x) <system-defined-code>
```

Each non-constant field definition is translated into two behavior definitions:

```
<name>() : <type> implementation fun(x) <system-defined-code>
set_<name>(<type>) : T_Unit
    implementation fun(x,y) <system-defined-code>
```

The **implementation** clauses that refer to fields are translated as follows:

```
<name> implementation <field-name>
```

is translated into

```
<name>(): <field-type> implementation fun() { return <field-name>; }
```

if **<field-name>** denotes a constant field and into

```
<name>(): <field-type> implementation fun() { return <field-name>; }
set_<name>(<field-type>) : T_Unit
    implementation fun(x) { set_<field-name>(x); }
```

if it denotes a non-constant field. For example, the (expanded) specification of **C_Ellipse** given above will be translated into

```
class C_Ellipse implements T_Ellipse {
    _center() : T_Point implementation <system-defined>;
    set__center(T_Point) : T_Unit implementation <system-defined>;
    _majorAxis() : T_Length implementation <system-defined>;
    set__majorAxis(T_Length) : T_Unit implementation <system-defined>;
```

```
        center() : T_Point implementation fun() { return _center(); };
        set_center(T_Point) : T_Unit
             implementation fun(x) { set__center(x); };
        majorAxis() : T_Length
             implementation fun() { return _majorAxis(); };
        set_majorAxis(T_Length) : T_Unit
             implementation fun(x) { set__majorAxis(x); };
        _minorAxis() : T_Length implementation <system-defined>;
        set__minorAxis(T_Length) : T_Unit implementation <system-defined>;
        minorAxis() : T_Length
             implementation fun() { return _minorAxis(); };
        set_minorAxis(T_Length) : T_Unit
             implementation fun(x) { set__minorAxis(x); };
};
```

Translation to types: Upon completion of the class expansion and field elimination discussed earlier, classes are translated into types. This translation proceeds by changing the keyword **selfclass** into **selftype**, changing **subclass of** and **implements** clauses into **subtype of** clauses, and changing each **class** specifier into a **type** specifier. Then, the standard type specification translation is performed with the following changes:

1. Instead of a **type** specification, an **concrete type** specification is produced;
2. The generation of reflexive constants is done according to the rules below, rather than according to the rules specified for type translation.

Note that there are no classes in the target language, only abstract and concrete types. Types in the target language that correspond to the classes in the source language have names of the form C_X in order to keep names consistent across the two languages. This is done to simplify typechecking; it does not have any effect on the semantics of the notions of type and class. For example, the class **C_Ellipse** defined above will be translated into

```
concrete type C_Ellipse;
C_Ellipse <= T_Ellipse;

association (C_Ellipse -> T_Point) _center <system-defined>;
association ((C_Ellipse, T_Point) -> T_Unit)
      set__center <system-defined>;
...
```

At this point, the classes have been translated into types. The only difference between types and classes remaining at this point is the difference between **concrete type** and **type** type specifiers and between **T_X** and **C_X** names.

Reflexive constants: Since classes, as well as types, can be used in expressions as objects, this translation step adds appropriate constant definitions for every class defined in the program. There are four cases:

Infinite classes If a class C_T implementing a type T_T has been specified as **infinite**, the following constant definition is added to the program:

```
const T_InfiniteClass(T_T) C_T;
```

An infinite class does not have an extent and can not create objects.

Finite classes If a class C_T implementing a type T_T has been specified as **finite**, the translation adds

```
const T_FiniteClass(T_T) C_T;
```

A finite class can not create objects, but it has an extent. The type **T_FiniteClass(X)** defines behavior **extent()→T_Set(X)** that returns a set of objects (the extent) of the class.

Manual classes Classes specified as **manual** can create objects and do not maintain extent. For each manual class C_T implementing a type T_T, the following definitions are generated:

```
const T_ManualClass(T_T) C_T;

association (T_ManualClass(T_T),
                   const-field-1-type,...,const-field-n-type) -> C_T
             new_C_T <system-defined>;
```

where each **const-field-k-type** is a type assigned in the class specification to the k-th constant field (including inherited fields). The behavior defined here creates a new object of class C_T. For example, the class specifications

```
manual class C_Person implements T_Person {
      const T_Natural _age;
};
manual class C_Child subclass of C_Person implements T_Child {
      const T_Set(T_Toy) _favoriteToys;
};
```

will eventually generate the definitions

```
const T_ManualClass(T_Person) C_Person;

association (T_ManualClass(T_Person), T_Natural) -> C_Person
      new_C_Person <system-defined>;

const T_ManualClass(T_Child) C_Child;

association (T_ManualClass(T_Child),
                   T_Natural, T_Set(T_Toy)) -> C_Child
      new_C_Child <system-defined>;
```

Regular classes If a class is specified as a **regular** class or without an explicit specifier, the definitions being generated are the same as for the manual class. The only change is in the name of the parametric type being assigned to the class constant, namely T_Class(T_T) instead of T_ManualClass(T_T). Regular classes maintain extents *and* are capable of object creation, therefore the type T_Class(X) is a subtype of both T_FiniteClass(X) and T_ManualClass(X). The class type hierarchy is depicted in Figure 3.5.

Function code translation In addition to the translation of type, behavior, and class specifications, the translation of function code also has to be performed. Translation of all interesting features will be presented here; translation of standard language operators is trivial and is therefore omitted.

Return elimination: If the function code has the form

```
<expr1> return <expr2>
```

it is translated into

```
<expr1>; <expr2>
```

Otherwise (no explicit return) it has the form ⟨**expr**⟩ and is translated into

```
<expr>; unit
```

where **unit** is a predefined object of type T_Unit

Operator elimination: All operators present in the code are translated into their equivalent behavioral form with an explicit receiver. For example, the code

```
a + b / c
```

will be translated into

```
a.plus(b.divide(c))
```

Local variables: Every local variable declaration

```
<type> <name>;
```

is transformed into

```
let T_Var(<type>) <name> = new C_Var(<type>) in
    { ..the rest of the function code up to the closing }.. }
```

Also, every occurrence of a local variable name `x` in the simple assignment context `x := <expr>` is translated into `x.set(<expr>)`, while every occurrence *outside* of the simple assignment context is translated into `x.get()`. Thus, the code

```
T_Integer k, l;
k := k.add(l);
```

will be translated into

```
let T_Var(T_Integer) k = new C_Var(T_Integer) in
  { k.set( (k.get()).add(l.get()) ); }
```

Assignment elimination: Simple assignments of the form

```
<name> := <expr>
```

are transformed into the equivalent behavioral form

```
set_<name>(<expr>)
```

All the other assignments have the form

```
<expr1>.<name>[<args>] := <expr2>
```

and are transformed into the equivalent behavioral form

```
<expr1>.set_<name>(<args>, <expr2>)
```

Since all local variable assignments have already been eliminated, all the simple assignments left are assignments to settable behaviors of the implicit receiver. The implicit receiver will be made explicit later on in the translation process.

If there are any assignments left at this stage, the program is invalid and is rejected. For example, during assignment elimination the following code

```
aString.substr(k,l) := aString2;
age := k.add(l);
```

is translated into

```
aString.set_substr(k,l,aString2);
set_age(k.add(l));
```

Explicit receiver: Every function that participates in an association is transformed to take its explicit receiver as the first argument. The identifier `self` is used to represent the receiver. In the function code, all behavior names `<name>` that are neither preceded by a dot nor appear in the construct `protect(<name>)` are replaced by `self.<name>`. Names that appear as `protect(<name>)` are replaced by `<name>`. The `protect` construct is designed to protect behaviors from being interpreted as behavior applications to the implicit receiver. For example, function

```
fun(x) { return fun(r) { return age.minus(r).minus(x) ; }; }
```

will be transformed into

```
fun(self, x) {
          return fun(r) { return  self.age.minus(r).minus(x); }; }
```

provided it is a part of an association. On the other hand, the function

```
fun() { return map(protect(negate)); }
```

will be transformed into

```
fun(self) { return self.map(negate); }
```

rather than into

```
fun(self) { return self.map(self.negate); }
```

because of the presence of `protect`.

Multiple argument elimination: Functions are made to accept a single argument of a product type rather than multiple arguments. This transformation is done to simplify type-checking. It is possible because both the source and the target language have multiple dispatch. The code is transformed in the following manner: all definitions

```
fun(a1,..,aN) { <expr> }
```

where `N` > 1 are translated into

```
fun(@x) { let a1 = @x_1 in let a2 = ... let aN = @x_N in <expr> }
```

where `@x` is a fresh variable. For example, the code

```
fun(self, x) {
        return fun(r) { return  self.age.minus(r).minus(x); }; }
```

is translated into

```
fun(@x) { let self = @x_1 in let x = @x_2 in
     return fun(r) { return  self.age.minus(r).minus(x); }; }
```

Object creation: Object creation in the source language is done by using operator `new` on classes. The translation process transforms the operator `new`, whose syntax is given below,

```
<new-operator> ::= "new" <class-name> ["(" [<initializers>] ")"]
<initializers> ::= <initializer> {"," <initializer>}*
<initializer> ::= <name> "=" <expr>
```

into a series of behavior applications in the following manner. First, the template for the code is generated:

```
{let @n = <class-name>.new_<class-name>(<args>) in {<tail>; @n}}
```

where `@n` is a fresh variable, and the meaning of `<args>` and `<tail>` is defined below. For each initializer `<name> = <expr>` used in a given `new` operator, two cases are considered:

1. There is a constant (immutable) field `<name>` defined on or inherited by the class `<class-name>`. Then, the expression `<expr>` is added to `<args>` in the position corresponding to the number assigned to the field by the class translation process described earlier.
2. Otherwise, the expression `@n.set_<name>(<expr>);` is added to the `<tail>`.

Consider the specification

```
class C_Circle implements T_Circle
    T_Point _center;
    const T_Length _majorAxis;
class C_Ellipse implements T_Ellipse extends C_Circle
    const T_Length _minorAxis;
```

and the code

```
new C_Ellipse(_center = aPoint,
                 _majorAxis = axis1, _minorAxis = axis2)
```

It will be translated into

```
{let @n = C_Ellipse.new_C_Ellipse(axis1,axis2) in
                       {@n.set__center(aPoint); @n}}
```

since the field `_center` is updatable, while the other fields are not.

typeif expressions: These expressions are designed for type-safe reflection and have a (simplified) syntax:

```
<typeif-expr> ::=
        "typeif" "(" <name> "is" <type> ")" "then" <expr> ";"
```

They check if the run-time of `<name>` is a subtype of `<type>` and execute the expression `<expr>` only if it is. In the expression `<expr>` it can therefore be assumed that the type of `<name>` is at least `<type>`. This last feature makes the **typeif** expressions special in terms of their typing.

During the translation, for each **typeif** expression, the behavior association

```
association T_Object -> <type> @as_<type> fun(x) <system-defined>
```

is added to the target language program, and the expression itself is transformed into

```
let <name> = <name>.@as<type> in {
   if <name>.notEqual(UNDEFINED) then { <expr> } }
```

The behavior `@as_<type>` has the following semantics: it returns an object `UNDEFINED` (of type `T_Null`) if its receiver has a type that is a subtype of `<type>`, and returns the receiver otherwise. Therefore, its result type is always at least `<type>`.

For example, as a result of the translation of expression

```
typeif ( anObject is T_Person ) then { anObject.age; };
```

the association

```
association T_Object -> T_Person @as_T_Person fun(x) <system-defined>
```

will be added to the program, while the expression itself will be translated into

```
let anObject = anObject.@as_T_Person in {
   if (anObject.notEqual(UNDEFINED)) then { anObject.age; }; };
```

Dot elimination: Finally, the dot notation used in the code so far is dropped in favor of functional notation. The behavior application expressions of the form

```
<expr0> "." <exprB> [ "(" <expr1> "," ... "," <exprN> ")" ]
```

are transformed into

```
<exprB> "(" <expr0> "," <expr1> "," ... "," <exprN> ")"
```

For example, the expressions

```
aString.substring(5, 6)
aPerson.age
```

are transformed into

```
substring(aString, 5, 6)
age(aPerson)
```

Final touch: The final step of the translation from the source to the target language consists of adjustment of the global program structure. First, type, subtype, behavior, and constant definitions of the basic system are added. Second, all definitions are grouped together in the target program sections: types, subtypes, constants, behaviors, and associations. Third, duplicate definitions are discarded (two definitions are duplicates of each other if they are identical up to whitespace elimination and free variable renaming). Finally, the named definitions are analyzed. If there are two different definitions for the same name, it is an error *unless* both conflicting definitions are behavior definitions.

In this section, the target language has been described. The target language presented is a simplified version of the full language that is specifically tailored for typechecking purposes. Then, the translation from the source language into the target language was described and illustrated. In the next section, typechecking algorithms for the target language will be presented.

4.2 Consistency

Typechecking a program in the target language, though significantly simpler compared to a source language program, is still a complicated task. There are several aspects to the issue of correctness of a target language program that will be considered in this section. Some of those aspects are *local* in that they can be verified by considering a single definition; others are *global* and deal with interactions between various definitions given in the program.

The entailment algorithm described in Section 4.3 forms the basis of the algorithms presented here; in a sense, target language program typing is simplified even further before this core algorithm is used. This section will provide the description of this simplification process.

This section is organized as follows. First, a formal notation for reasoning about types and certain basic notions (such as types, constrained types, entailment, and variance annotations) are introduced in Section 4.2.1. The remaining sections describe various steps of program verification in the order they are to be performed. First, local monotonicity checking is described in Section 4.2.2. Then the set of type specifications given by the user is checked for acyclicity and complexity. This process is described in Section 4.2.3. After these conditions are checked, type expansions (which is a form of type expression simplification) can be performed. This is described in Section 4.2.4. After this task is completed, the main algorithms of Section 4.3 can be applied. These algorithms are used in all subsequent steps of the verification process. Section 4.2.5 describes verification of the user-supplied constraints. After these are verified, global behavior specification consistency is tested. This step is described in Section 4.2.6. The next step, described in Section 4.2.7, performs type derivation and checks consistency of function and program bodies with respect to their specifications. This is where most of the typechecking occurs. Finally, on the last step, the dispatch consistency (absence of "message not understood" and "message ambiguous" errors) is verified. This last step is described in Section 4.2.8.

4.2.1 Notation

Formal reasoning about types has an established notation. For instance, [AC96, BM96a, GM96, QKB96, AW93], as well as many others use notations based on the common principles that will be described in this section.

This section presents an informal overview of the theoretical notions behind the typechecking algorithms. It gives sufficient information to implement and use the typechecking algorithms presented in this chapter. The formal treatment of all the concepts presented here will be given in Chapter 5.

Type constructors are represented by either lowercase Latin letters $a, b, \ldots$ or by type names **T_A**, **T_B**, ... Type variables are represented by Greek letters $\alpha, \beta, \gamma, \ldots$ and arbitrary type expressions are represented by uppercase Latin letters $A, B, C, \ldots$. The notation $A[\vec{\alpha}]$ is used to denote a type expression with free variables $\alpha_1, \alpha_2, \ldots, \alpha_n$, and the notation $A[\vec{B}/\vec{\alpha}]$ denotes a type expression A in which type expressions B_i have been substituted for variables α_i. The shortcut $(A_1, \ldots, A_n)$ will be used for product types **T_Product**$_n(A_1, \ldots, A_n)$, and the shortcuts $A \rightarrow R$ and $A \rightarrow_b R$ will be used for functional and behavior types **T_Function**(A, R) and **T_Behavior**(A, R), respectively. The type operators lub(...) and glb(...) will be used to denote the least upper bound and the greatest lower bound of a given set of types. The syntax of type expressions is given below:

$$A ::= a \mid \alpha \mid a(A_1, \ldots, A_n) \mid A_1 \rightarrow A_2 \mid A_1 \rightarrow_b A_2 \mid (A_1, \ldots, A_n) \mid \mathrm{lub}(A_1, \ldots, A_n) \mid \mathrm{glb}(A_1, \ldots, A_n)$$

The type $a(A_1, \ldots, A_n)$ is considered *valid* iff a has arity n. The type expression a is a shortcut for $a()$ and thus requires a to be of arity 0.

For example, both $A = a(b(\alpha, d), \alpha \rightarrow \alpha)$ and $B = d(b, a(c))$ are type expressions. A has a set of free variables $FTV(A) = \{\alpha\}$ and is *open*, while B has an empty set of free variables and is therefore *closed*. The term *primary functor* (*pfunctor*) will be used to denote the outermost type constructor of a simple type. For example, $pfunctor(A) = a$ and $pfunctor(B) = d$.

In the theoretical part of this dissertation, closed types are modelled as *regular trees* over the alphabet defined by the set of all type constructors plus the type operators lub and glb. Intuitively, a regular tree is a closed, possibly infinite, type expression that can be described in a finite form. For every closed (ground) type there is a corresponding regular tree that represents it. An example of an infinite regular tree is a solution of the type equation $\alpha = b \rightarrow \alpha$, which is an infinite closed type expression $b \rightarrow b \rightarrow b \rightarrow \ldots$ There is a subtyping relationship (denoted $\preceq$) defined on regular trees (defined in Chapter 5) by the relationships specified by the user in the user type graph $\mathcal{G}$ (see Section 3.1.7, page 62; see also Definition 4.6 below). Subtyping is a partial order over the domain of ground types.

Constraints and entailment

Constraints are inequalities between type expressions. For example,

$$a(\alpha) \preceq a(\beta)$$

is a constraint. Constraint satisfiability is defined over the domain of regular trees described earlier. Namely, a constraint is ***satisfiable*** if there is a set of regular trees (valuation) such that when these regular trees are substituted for variables, the constraint in question becomes true. A special case of a constraint is a constraint denoted as $TRUE(\alpha_1, \ldots, \alpha_n)$ which is considered to be satisfied for all values of $\alpha_1, \ldots, \alpha_n$.

A *constraint set*, or a constraint conjunction, is defined as a conjunction of a finite number of constraints. Satisfiability of a constraint set is defined as existence of a valuation that turns the constraint conjunction into a true formula. $\texttt{T\_Integer} \preceq \alpha \rightarrow \beta$ is an example of an unsatisfiable constraint, while

$$\alpha \preceq \beta \wedge \beta \preceq \texttt{T\_Integer}$$

is an example of a satisfiable constraint set (conjunction). Constraint sets will be denoted $\mathbb{C}$ in the rest of this dissertation.

Entailment is a relationship between two constraint sets defined as follows:

$$(\mathbb{C}_1 \vdash \mathbb{C}_2) \overset{\text{def}}{\Longleftrightarrow} (\forall \vec{\alpha}_1 \ (\mathbb{C}_1[\vec{\alpha}_1] \Rightarrow \exists \vec{\alpha}_2\colon \ \mathbb{C}_2[\vec{\alpha}_1, \vec{\alpha}_2]))$$

In other words, a constraint set entails another if for each valuation of the first set that turns it into a true statement there exists a valuation of the second that turns it into a true statement such that these two valuations give the same values to variables that participate in both of them.

The relationship $\vdash$ can be intuitively understood as follows. If there are two sets of constraints, $\mathbb{C}_1$ and $\mathbb{C}_2$, then $\mathbb{C}_1 \vdash \mathbb{C}_2$ is true if and only if the second constraint set follows from the first. The statement $\vdash \mathbb{C}$ therefore has a meaning that $\mathbb{C}$ is satisfiable. Use of subscript $\mathcal{G}$ in $\vdash_{\mathcal{G}}$ means that the entailment is checked in the context of the type system defined by the user type graph $\mathcal{G}$.

For example, $\alpha \preceq \beta, \beta \preceq \gamma \vdash \alpha \preceq \gamma$, but $\alpha \preceq \beta \nvdash \beta \preceq \alpha$. Another example of true entailment is

$$\vdash \alpha \preceq \texttt{T\_Integer}$$

as it states that there exists a type α that is a subtype of **T_Integer**. However, the formula

$$TRUE(\alpha) \vdash \alpha \preceq \texttt{T\_Integer}$$

is false since it states that any type α is a subtype of **T_Integer** which is obviously false.

The verification algorithm for $\vdash_{\mathcal{G}}$ will be given in Section 4.3.

The notion of entailment introduced above is central to the typechecking mechanism presented in this dissertation. All typechecking tasks are ultimately formulated in terms of entailment.

Constrained types

The types and constraints described above are used in the construction of *constrained types*. A simple constrained type has the following form:

$$\forall \alpha_1, \ldots, \alpha_n \ (\mathbb{C}).A$$

where $\mathbb{C}$ is a constraint set, A is a type, and $\alpha_1, \ldots, \alpha_n$ are type variables free in A and $\mathbb{C}$. The syntax of simple constrained types is as follows:

$$X ::= \forall \alpha_1, \ldots, \alpha_n\ (\mathbb{C}).A$$
$$\mathbb{C} ::= A_1 \preceq A_2 \mid \mathbb{C} \wedge A_1 \preceq A_2$$

We will often omit the quantification and write $(\mathbb{C}).A$ instead of $\forall \alpha_1, \ldots, \alpha_n\ (\mathbb{C}).A$. When the constraints are empty, they will also be omitted. For example, both $X = b$ and $Y = (b \preceq \alpha).\alpha$ are simple constrained types. Their unabbreviated form is $().b$ and $\forall \alpha\ (b \preceq \alpha).\alpha$, respectively.

The intuitive meaning of a simple constrained type $\forall \alpha_1, \ldots, \alpha_n\ (\mathbb{C}).A$ can be understood as follows. Let S be the set of all closed types $A[\vec{T}/\vec{\alpha}]$ such that T_i are closed types satisfying the subtyping constraints $\mathbb{C}[\vec{T}/\vec{\alpha}]$. Then, the simple constrained type $\forall \alpha_1, \ldots, \alpha_n\ (\mathbb{C}).A$ denotes the set of all greatest lower bounds of S. For example, consider the simple constrained type $\forall \alpha\ (b \preceq \alpha).\alpha$. The set S is the set of all types that are supertypes of b. The greatest lower bound of S is b and therefore $\forall \alpha\ (b \preceq \alpha).\alpha \equiv b$. This can be directly verified by using Definition 4.1.

Subtyping defined on ground types induces subtyping on simple constrained types defined as follows.

Definition 4.1 (Subtyping of simple constrained types).

$$(\forall \vec{\alpha}_1\ (\mathbb{C}_1).A_1 \preceq \forall \vec{\alpha}_2\ (\mathbb{C}_2).A_2) \overset{\text{def}}{\Longleftrightarrow} \forall \vec{\alpha}_2\ (\mathbb{C}_2 \Rightarrow \exists \vec{\alpha}_1\ \mathbb{C}_1 \wedge A_1 \preceq A_2)$$

where the quantification is over the domain of ground types. □

This defines subtyping as a partial order over the domain of simple constrained types factored by the equivalence relationship $\equiv$ defined as $(X \equiv Y) \overset{\text{def}}{\Longleftrightarrow} (X \preceq Y \wedge Y \preceq X)$. Intuitively, the equivalence relationship allows one to identify syntactically different, but semantically equivalent simple constrained types, like $\forall \alpha\ (b \preceq \alpha).\alpha$ and $().b$ considered above.

The following is the intuitive meaning of the above subtyping definition. If X_1 and X_2 are simple constrained types denoting respectively sets S_1 and S_2, then $X_1 \preceq X_2$ iff for each element s_2 of S_2 there is an element s_1 of S_1 such that $s_1 \preceq s_2$. For example, consider the simple constrained type

$$X_f = \forall \alpha\ (\alpha \preceq \texttt{T\_Integer}).(\alpha \rightarrow \alpha)$$

X_f denotes all function types that accept an argument of any type which is a subtype of **T_Integer** and produce the result of the same type as the argument supplied. What is the relationship between X_f and the function type $X_i = \texttt{T\_Integer} \rightarrow \texttt{T\_Integer}$? The type X_i corresponds to a singleton set consisting of **T_Integer→T_Integer** itself. This element belongs also to the set denoted by X_f and therefore $X_f \preceq X_i$, but $X_f \not\succeq X_i$. In order to see why this is the case, consider using a function f of type X_f where X_i is expected. f accepts arguments of type **T_Integer**, and produces the result of type at least **T_Integer**, and thus conforms to the specification given by X_i. On the other hand, an attempt to use a function i of type X_i where X_f is expected leads to a type error, since X_f not only requires its result type to be **T_Integer**, but also to be of at least the same type as its argument. The specification X_i guarantees the first of these two requirements, but not the second.

Note that if $\mathbb{C}$ is an unsatisfiable set of constraints, then $X \preceq (\mathbb{C}).A$ for any X and A. Thus constrained types with unsatisfiable constraint sets play the role of the top type in the hierarchy (denoted $\top$). Types of the form $().\alpha$, on the other hand, play the role of the bottom type (denoted $\perp$).

The notion of a primary functor is generalized for simple constrained types. The primary functor of a simple constrained type $X = (\mathbb{C}).A$ is defined to be the primary functor of A.

Note that the notation used for types and simple constrained types introduced in this section is parallel to the notation used to specify types, subtypes, and constraints in the target language (Section 4.1.1). This property of the target language will enable the description of consistency conditions directly in terms of the notation defined here, with no additional translation steps.

Simple constrained types are further generalized to *constrained types*. A constrained type has a form $X = \text{glb}_i X_i$, where X_i are simple constrained types. The intuitive meaning is that if each of X_i denotes a set of types S_i, then X denotes a set of types S which is the set of all lower bounds of the union $\bigcup_i S_i$. Subtyping is also trivially generalized for the constrained types as given in the following definition.

Definition 4.2 (Subtyping of constrained types).

$$(\text{glb}_i X_i^1 \preceq \text{glb}_j X_j^2) \overset{\text{def}}{\Longleftrightarrow} (\forall j \exists i: \quad X_i^1 \preceq X_j^2)$$

□

For example,

$$\text{glb}(\texttt{T\_Integer}\to\texttt{T\_Integer}, \forall\alpha\ (\texttt{T\_Real} \preceq \alpha).(\alpha\to\alpha)) \preceq \texttt{T\_Real}\to\texttt{T\_Real}$$

even though

$$\texttt{T\_Integer}\to\texttt{T\_Integer} \not\preceq \texttt{T\_Real}\to\texttt{T\_Real}$$

This generalization is useful in describing behavior types for behaviors that have several types (signatures) defined for them. For example, **cat** (concatenation) can have both signatures

$$(\texttt{T\_List}(\alpha), \texttt{T\_List}(\alpha))\to\texttt{T\_List}(\alpha)$$

and

$$(\texttt{T\_String}, \texttt{T\_String})\to\texttt{T\_String}$$

Then the type of **cat** can be precisely described by the constrained type

$$\text{glb}((\texttt{T\_List}(\alpha), \texttt{T\_List}(\alpha))\to\texttt{T\_List}(\alpha), (\texttt{T\_String}, \texttt{T\_String})\to\texttt{T\_String})$$

Constrained types will be denoted by the uppercase Latin letters $X, Y, Z, W, \ldots$ in the rest of this dissertation.

Variance annotations

Variance annotations are variable annotations that are drawn from the set $\{-, +, 0\}$. An annotated type expression is a type expression with variables annotated by one of $+$ (covariance), $-$ (contravariance), or 0 (novariance). For example, $a(b, c(\alpha^0))$ is an annotated type expression.

If a particular occurrence of a variable in a type expression is annotated by $+$ ($-$, 0), it means that the type expression changes covariantly (contravariantly, novariantly) with types substituted for that occurrence of the variable. For example, consider a type expression $A[\alpha]$ that consists of just a variable α, i.e. $A[\alpha] = \alpha$. Let $\underline{t}$ and $\overline{t}$ be some types. When we use $\underline{t}$ for α in $A[\alpha]$, we obtain $\underline{A} = \underline{t}$; when we use $\overline{t}$, we obtain $\overline{A} = \overline{t}$. Since

$$\underline{t} \preceq \overline{t} \Rightarrow \underline{A} \preceq \overline{A}$$

the only occurrence of α in $A[\alpha]$ is covariant and should be annotated by $+$: $A[\alpha] = \alpha^+$. A more complicated example is the type expression $B[\alpha] = \alpha\to\alpha$. Here the two occurrences of α have different variances. Namely, consider the first occurrence. $\underline{B} = \underline{t}\to\alpha$ and $\overline{B} = \overline{t}\to\alpha$ and therefore

$$\underline{t} \preceq \overline{t} \Rightarrow \overline{B} \preceq \underline{B}$$

Thus, the first occurrence of α in $B[\alpha]$ is contravariant. Consider the second occurrence. In this case, $\underline{B} = \alpha\to\underline{t}$ and $\overline{B} = \alpha\to\overline{t}$ and therefore

$$\underline{t} \preceq \overline{t} \Rightarrow \underline{B} \preceq \overline{B}$$

Table 4.1: Variance combination

$\odot$	+	-	0
+	+	-	0
-	-	+	0
0	0	0	0

Thus, the second occurrence is covariant. Therefore, the annotated expression will be $B[\alpha] = \alpha^{-}\rightarrow\alpha^{+}$

The following algorithm formalizes the process of annotating a given type expression. Given a type, its *canonical annotated form* is defined as follows.

Definition 4.3 (Canonical annotated form (CAF)). Given a type expression A, its canonical annotated form is given by the annotation algorithm given below, with the call $annotate(A, +)$.

$$annotate(A, v) = \begin{cases} a & \text{if } A = a \\ \alpha^{v} & \text{if } A = \alpha \\ a(A'_1, \dots, A'_n) & \text{if } A = a(A_1, \dots, A_n) \\ \text{where } A'_i = annotate(A_i,\ Var(a, i) \odot v) & \\ \mathsf{lub}(A'_1, \dots, A'_n) & \text{if } A = \mathsf{lub}(A_1, \dots, A_n) \\ \text{where } A'_i = annotate(A_i, v) & \\ \mathsf{glb}(A'_1, \dots, A'_n) & \text{if } A = \mathsf{glb}(A_1, \dots, A_n) \\ \text{where } A'_i = annotate(A_i, v) & \end{cases}$$

Here, $Var(a, i)$ is defined according to the predefined variance of the i-th parameter of the type constructor a as follows: it is $+$ if the parameter is defined as covariant, $-$ if it is defined as contravariant, and 0 if it is defined as novariant. The commutative binary operation $\odot$ on variances is defined by Table 4.1. □

For example, the CAF of the expression $(a, b, (\alpha\rightarrow\beta\rightarrow\alpha))$ is

$$\begin{aligned} & annotate((a, b, (\alpha\rightarrow\beta\rightarrow\alpha)), +) \\ & \quad = (annotate(a, + \odot +), annotate(b, + \odot +), annotate(\alpha\rightarrow\beta\rightarrow\alpha, + \odot +)) \\ & \quad = (annotate(a, +), annotate(b, +), annotate(\alpha\rightarrow\beta\rightarrow\alpha, +)) \\ & \quad = (a, b, annotate(\alpha, - \odot +)\rightarrow annotate(\beta\rightarrow\alpha, + \odot +) \\ & \quad = (a, b, annotate(\alpha, -)\rightarrow annotate(\beta\rightarrow\alpha, +) \\ & \quad = (a, b, \alpha^{-}\rightarrow annotate(\beta, - \odot +)\rightarrow annotate(\alpha, + \odot +)) \\ & \quad = (a, b, \alpha^{-}\rightarrow\beta^{-}\rightarrow\alpha^{+}) \end{aligned}$$

since the first argument of $\rightarrow$ is defined as contravariant $(-)$, while the second one is defined as covariant $(+)$; at the same time, all arguments of a product type $(\dots)$ are defined as covariant.

Once a CAF C of a type is computed, its *variance set* V is defined as a set of tuples of the form $\langle\alpha, S\rangle$, where α is a type variable and S is a set of all variances assigned to α in C. S can be also thought of as a map from a variable name to a set of all its annotations in a given expression. For example,

$$V((a, b, (\alpha\rightarrow\beta\rightarrow\alpha))) = V((a, b, \alpha^{-}\rightarrow\beta^{-}\rightarrow\alpha^{+})) = \{\langle\alpha, \{-, +\}\rangle, \langle\beta, \{-\}\rangle\}$$

There is a partial order (*strength*) defined on variance sets. A variance set V_1 is *stronger* then a variance set V_2 (denoted $V_1 \gg V_2$) iff for each type variable α such that $\langle\alpha, S_2\rangle \in V_2$ there exists $\langle\alpha, S_1\rangle \in V_1$ such that either $S_2 \subseteq S_1$, or $0 \in S_1$, or $\{+, -\} \subseteq S_1$.

For example,

$$\{\langle\alpha,\{0\}\rangle,\langle\beta,\{-,+\}\rangle,\langle\gamma,\{-\}\rangle,\langle\kappa,\{+\}\rangle\} \gg \{\langle\alpha,\{+,-\}\rangle,\langle\beta,\{0\}\rangle,\langle\gamma,\{-\}\rangle\}$$

but

$$\{\langle\alpha,\{-\}\rangle\} \not\gg \{\langle\alpha,\{0\}\rangle\}$$

The "strength" relationship defined above has the following meaning. Assume that there are two type expressions, A and B, that share some type variables $\alpha_1,\ldots,\alpha_n$. Then, if the variance set V_A of A is stronger then the variance set V_B of B, the following holds:

$$A[\vec{r}'/\vec{\alpha}] \preceq A[\vec{r}/\vec{\alpha}] \Rightarrow B[\vec{r}'/\vec{\alpha}] \preceq B[\vec{r}/\vec{\alpha}] \qquad (4.1)$$

for all closed type expressions $r_1,\ldots,r_n,r'_1,\ldots,r'_n$. For example, let $\mathtt{T\_List}(X)$ be a novariant type. Let $A = \mathtt{T\_List}(\alpha)$ and $B = \alpha\to\alpha$. Then $V_A = \{\langle\alpha,\{0\}\rangle\}$ and $V_B = \{\langle\alpha,\{+,-\}\rangle\}$ and therefore $V_A \gg V_B$. Let us check the property (4.1). Indeed,

$$\begin{aligned} A[\vec{r}'/\vec{\alpha}] \preceq A[\vec{r}/\vec{\alpha}] &\equiv \mathtt{T\_List}(r') \preceq \mathtt{T\_List}(r) \\ &\equiv r' = r \\ &\Rightarrow (r'\to r') = (r\to r) \\ &\Rightarrow (r'\to r') \preceq (r\to r) \\ &\equiv B[r'/\alpha] \preceq B[r/\alpha] \end{aligned}$$

On the other hand, if we take $A = \alpha$, then $V_A = \{\langle\alpha,\{+\}\rangle\}$ and $V_A \not\gg V_B$. In this case, $A[\vec{r}'/\vec{\alpha}] = r'$ and $A[\vec{r}/\vec{\alpha}] = r$. The property (4.1) is then formulated as

$$r' \preceq r \Rightarrow (r'\to r') \preceq (r\to r)$$

which is false (e.g., take $r' = \mathtt{T\_Integer}$ and $r = \mathtt{T\_Real}$).

The notion of a *canonical annotated form* is generalized for simple constrained types. The canonical annotated form for a simple constrained type $X = (\mathbb{C}).A$ is defined as follows:

Definition 4.4 (Canonical annotated form (CAF) for simple constrained types).

$$\begin{aligned} &\mathrm{CAF}((L_1 \preceq U_1,\ldots,L_n \preceq U_n).A) \overset{\mathrm{def}}{=} \\ &\qquad (\mathit{annotate}(L_1,+) \preceq \mathit{annotate}(U_1,-),\ldots, \\ &\qquad\qquad \mathit{annotate}(L_n,+) \preceq \mathit{annotate}(U_n,-)).\mathit{annotate}(A,+) \end{aligned}$$

□

For example, the CAF of the constrained type $(\alpha\to\beta \preceq \alpha).\beta\to\alpha$ is $(\alpha^-\to\beta^+ \preceq \alpha^-).\beta^-\to\alpha^+$. The notion of a variance set is trivially generalized to constrained types as well.

The above definition states that expressions on the left side of the subtyping relationship are considered to be in covariant positions, while those on the right are considered to be in contravariant positions. The following is the intuitive reason for this to be true. Let $C = \alpha \preceq \beta$ be a constraint, and let t_a and t_b be such types that $C[t_a/\alpha,t_b/\beta]$ is satisfied. Then

$$(t'_a \preceq t_a \wedge t_b \preceq t'_b) \Rightarrow C[t'_a/\alpha,t'_b/\beta]$$

In other words, a constraint remains true when the expression on the left changes covariantly and the expression on the right changes contravariantly.

Conclusion

In this section, the notation for dealing with types, constrained types, and variance annotations has been established. In the following sections, this notation will be used to establish consistency conditions placed on the type system.

In the following discussion, the concrete type specifiers will be omitted, as they are not relevant to type definition consistency. Also, associations will be treated like behavior definitions (their function part will be ignored). Function associations are only relevant to Section 4.2.7 and Section 4.2.8, while the distinction between abstract and concrete types is only relevant to Section 4.2.8.

4.2.2 Local monotonicity

All type and behavior definitions present in the program are required to be *locally monotonic*. Local monotonicity corresponds to the intuition that a parametric type (behavior) definition has to ensure that the type (behavior) being defined maintains its validity in the presence of covariant changes in its parameters. The conditions for local monotonicity are outlined below.

Definition 4.5 (Local monotonicity).

1. A type definition of the form

$$\mathbf{type}\ (\mathbb{C})\ a(\alpha_1^{v_1}, \ldots, \alpha_n^{v_n})$$

   is *locally monotonic* iff

$$\{\langle \alpha_i, \{v_i\} \rangle\} \gg V(\mathbb{C})$$

2. A behavior definition of the form

$$\mathbf{behavior}(\mathbb{C})\ A \rightarrow R\ \langle \mathbf{name} \rangle;$$

   is *locally monotonic* iff

$$V(A) \gg V(\mathbb{C}) \cup V(R)$$

□

For example, consider the type specification

```
type (X <= T_Ordered(X)) T_OrderedSet(covar X);
```

Its constraint set is $\mathbb{C} = \alpha \preceq \mathbf{T\_Ordered}(\alpha)$. The CAF of the constraint set is $\mathbb{C}' = \alpha^+ \preceq \mathbf{T\_Ordered}(\alpha^{(-\odot-)})$, i.e. $\alpha^+ \preceq \mathbf{T\_Ordered}(\alpha^+)$. The variance set of constraints is therefore $V(\mathbb{C}) = \{\langle \alpha, \{+\} \rangle\}$. On the other hand, the variance set of the body is $V = \{\langle \alpha, \{+\} \rangle\}$ and therefore $V \gg V(\mathbb{C})$. Therefore the type specification under consideration is monotonic.

On the other hand, the type specification

```
type (T_Natural <= X) T_SpecialSet(covar X);
```

is not monotonic, since CAF of the constraints is $\mathbb{C}' = \mathbf{T\_Natural} \preceq \alpha^-$, the variance set of constraints is $V(\mathbb{C}) = \{\langle \alpha, \{-\} \rangle\}$, the variance set of the body is $V = \{\langle \alpha, \{+\} \rangle\}$, and $V \not\gg V(\mathbb{C})$. Therefore, the type definition above is rejected.

In the presence of type definitions

$$\mathbf{type}\ c(\alpha^+);$$
$$\mathbf{type}\ n(\alpha^0);$$

the behavior definition

$$\texttt{behavior}\ c(\alpha) \to \alpha\ \texttt{pick};$$

will be monotonic:

$$V(A) = V(c(\alpha)) = \{\alpha, \{+\}\} \gg \{\alpha, \{+\}\} = \emptyset \cup V(\alpha)$$

while the behavior definition

$$\texttt{behavior}\ \alpha \to n(\alpha)\ \texttt{createRef};$$

will not be considered monotonic:

$$V(A) = V(\alpha) = \{\alpha, \{+\}\} \not\gg \{\alpha, \{0\}\} = \emptyset \cup V(n(\alpha))$$

and will therefore be rejected.

The local monotonicity condition is never used in the theory and can be lifted without impacting its correctness. The reason for introduction of this restriction is not a theoretical, but a practical one. Namely, while typechecking will work just as well in the presence of non-monotonic type specifications, interpreting its results would be a challenge. To illustrate this point, consider the following example. Let a type `T_List` of updatable lists be defined as

```
type T_List(novar X);
```

and let `inject` be the behavior with the following intended semantics: `inject` applied to an object o of any type X produces a list of type `T_List`(X) that contains a single element o. The specification of `inject` will therefore be as follows:

```
behavior X -> T_List(X) inject;
```

The type of `inject` will therefore be

$$T = \forall\alpha.(\alpha \to_b \texttt{T\_List}(\alpha))$$

which is not monotonic. The meaning of this type as interpreted by the type system is drastically different from what the user might expect. Namely, the type system assumes that `inject` has *all* the types denoted by T for different values of α. This means that a function that implements `inject` is required to produce an object of the type $R = \text{glb}_{\alpha \preceq X}\, \texttt{T\_List}(\alpha)$ for any receiver object of type X. Clearly, no object can simultaneously possess types `T_List`(α) for all α below X and therefore a function implementing `inject` can never be written. Thus, even though typechecking of expressions involving `inject` proceeds (almost) as expected, the assumptions that the type system makes about the meaning of the type T are counterintuitive at best. In order to avoid situations like these, the monotonicity restrictions are placed on all type specifications appearing in a program.

In this section, the concept of *local monotonicity* has been constructively defined. The local monotonicity definition is at the same time an algorithm for checking local monotonicity of type, behavior, and function definitions.

4.2.3 Simplicity and the user type graph

The type system and the systems of type constraints used in the program should define subtyping as a partial order. In addition to that, the defined order should be "simple enough" to guarantee the termination of the algorithms described in Section 4.3. In this section, it will be shown how the above conditions are formulated and verified.

In order to formulate and deal with these conditions, the notion of *the user type graph* $\mathcal{G}$ is introduced. It will be used later to define and check acyclicity and simplicity conditions.

Definition 4.6 (User type graph ($\mathcal{G}$)). $\mathcal{G}$ is a directed graph with labeled edges. Vertices in $\mathcal{G}$ are the type definitions of the given program. $\mathcal{G}$ has an edge from a to b iff there is a subtype definition of the form

$$(\mathbb{C})\ a(\vec{A}) \leq b(\vec{B})$$

in the program. The edge is labeled by the subtype definition above. □

Thus, the user type graph is a directed graph with labeled vertices and edges.

It is required that $\mathcal{G}$ be *acyclic*. This is a sufficient condition for the subtyping defined by the set of definitions to be a partial order, as will be shown in Chapter 5. For the theoretical treatment of types and subtyping this is all that is needed. However, this is insufficient to ensure termination of the core (entailment) algorithm used for type checking.

In order to see why this is the case consider the following definitions:

```
type T_List(novar X);
(T_SpecialList(X) <= T_List(X)) T_SpecialList(X) <= T_List(X);
```

What this definition is saying is that **T_SpecialList**(X) is a subtype of **T_List**(X) if **T_SpecialList**(X) is a subtype of **T_List**(X) — an apparent recursion. When an algorithm attempts to establish whether a given type **T_SpecialList**(A) is a subtype of another type **T_List**(A), it checks the condition first, thus going into infinite loop. Of course, this is a trivial example, but it does show that acyclicity of the type graph alone does not guarantee termination.

Therefore it is also required that $\mathcal{G}$ has a *valid ranking*. A valid ranking is informally an assignment of different positive "complexity" indices to all type constructors in the graph. One type constructor is "more complex" than the other if the second one can be used to test some subtyping hypothesis involving the first one. As long as this "more complex" relationship does not have any cycles in the graph, assignment of complexity indices is possible. Considering the above example with **T_List** and **T_SpecialList**, we can see that **T_List** is "more complex" than **T_SpecialList** since the later is required to verify the hypothesis **T_SpecialList**(A) $\preceq$ **T_List**(B). For the same reason, **T_SpecialList** is "more complex" than **T_List**, and therefore the type graph defined by this example does not have a valid ranking.

The formal definition of valid ranking as well as the algorithm that checks whether a given graph $\mathcal{G}$ has a valid ranking is given below.

Definition 4.7 (Ranking and valid ranking). *Ranking* is an assignment of positive integer numbers (ranks) $\bar{a}$ to all type constructors with non-zero arity a in $\mathcal{G}$. A *valid ranking* is a ranking such that

1. For each path P between two constructors a and b in the user type graph $\mathcal{G}$ the following condition holds:

$$\max(E \cup R) < \min(\{\bar{a}, \bar{b}\} \setminus I)$$

   Here, E is a union of *rank sets* for all edges in the path P, R is a union of *rank sets* for all vertices in the path P, and I is the set of all 0-ary type constructors in $\mathcal{G}$. It is assumed that $\max(\emptyset) = -\infty$ and $\min(\emptyset) = +\infty$. The operation $\setminus$ denotes set-theoretical difference.

2. For non-0-ary type constructor a in $\mathcal{G}$ the following condition holds:

$$\max(E_a) < \bar{a}$$

   where E_a is the rank set of the vertex a in $\mathcal{G}$.

A *rank set* for an edge is defined as a set of ranks of all non-0-ary type constructors that participate in the label of that edge in positions other than primary functors of the principal inequality. In other words, if the label of the edge is

$$(\mathbb{C})\ a(\vec{A}) \leq b(\vec{B})$$

then the rank set for this edge includes ranks of all non-0-ary type constructors participating in $\mathbb{C}, \vec{A}$, and $\vec{B}$.

A *rank set* for a vertex is defined as a set of ranks of all non-0-ary constructors participating in conditions placed on the label of the vertex. In other words, if a vertex a is labeled with

$$\textbf{type}\ (\mathbb{C})\ a(...);$$

then the rank set for this vertex will include ranks of all non-0-ary type constructors that participate in $\mathbb{C}$. □

Given a set of type and subtype definitions, the set of inequalities between constructor ranks that defines a valid ranking is constructed. Each inequality has the form

$$\max(\text{set of ranks } S_i) < \min(\{\bar{a}_i, \bar{b}_i\})$$

and can be represented as

$$\bigwedge_{s \in S_i} (s < \bar{a}_i \wedge s < \bar{b}_i)$$

A valid ranking is a ranking that satisfies all inequalities, i.e. satisfies the formula

$$\bigwedge_{i} \bigwedge_{s \in S_i} (s < \bar{a}_i \wedge s < \bar{b}_i) \qquad (4.2)$$

This formula has the form

$$\bar{a}_1 < \bar{a}_2 \wedge \bar{a}_3 < \bar{a}_4 \wedge ... \bar{a}_{n-1} < \bar{a}_n$$

where $\bar{a}_i$ are (not necessarily distinct) rankings. Such a conjunct defines a directed graph G with vertices $\bar{a}_i$ and edges $\bar{a}_i \rightarrow \bar{a}_j$ for all inequalities $\bar{a}_i < \bar{a}_j$ in the conjunct. If G is acyclic, there exists a ranking such that the formula is satisfied. Such ranking is a valid ranking.

Thus checking for the existence of a valid ranking is done as follows:

1. A formula of the form described in Equation 4.2 is constructed
2. The graph G is constructed and checked for acyclicity. If it is acyclic, succeed; otherwise, fail

As an example, consider the set of type and subtype definitions

```
type T_Printable;
type T_Set(covar X);
(X <= T_Printable) T_Set(X) <= T_Printable;
type T_OutputStream(contravar X);
type T_SetOS(novar X);
T_SetOS(X) <= T_Set(T_OutputStream(X));
```

For brevity, p will be used for **T_Printable**, s for **T_Set**, o for **T_OutputStream**, and q for **T_SetOS**. The above definition can be represented in theoretical notation as

type p;

type $s(\alpha^+)$;

$(\alpha \preceq p)\ s(\alpha) \preceq p$;

type $o(\alpha^-)$;

type $q(\alpha^0)$;

$q(\alpha) \preceq s(o(\alpha))$;

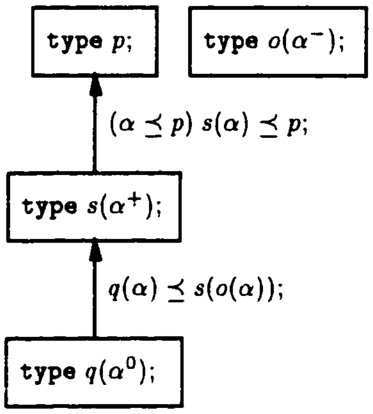
type p;
type $o(\alpha^-)$;
$(\alpha \preceq p)\ s(\alpha) \preceq p$;
type $s(\alpha^+)$;
$q(\alpha) \preceq s(o(\alpha))$;
type $q(\alpha^0)$;

Figure 4.1: Example user type graph $\mathcal{G}$

The user type graph $\mathcal{G}$ for this hierarchy is presented in Figure 4.1.

There are three possible paths in this graph: $s \to p$, $q \to s$, and $q \to p$. Only these paths contribute to the resulting formula since all vertex rank sets are empty. Edge rank sets are $\{\bar{p}\}$ for $s \to p$ and $\{\bar{s}\}$ for $q \to s$. The inequalities are as follows:

$$
\begin{array}{ll}
-\infty < \bar{s} & \text{for } s \to p \\
\bar{o} < \min(\bar{q}, \bar{s}) & \text{for } q \to s \\
\bar{o} < \bar{q} & \text{for } q \to p
\end{array}
$$

and the formula (Equation 4.2) is

$$\bar{o} < \bar{q} \wedge \bar{o} < \bar{s}$$

This fromula defines an acyclic graph G and therefore a valid ranking exists.

In this section, acyclicity and simplicity conditions have been described. The notions of *the user type graph* and *a valid ranking* were defined and used to formulate the above conditions. The algorithm for checking the simplicity condition has been presented and exemplified.

4.2.4 Type expansion

In order to perform the necessary checking, the concept of *type expansion* is used. Informally, a type expression is *expanded* if it explicitly lists all the constraints implicit in the definitions of types that participate in the expression. For example, if we have the definition

```
type (X <= T_Person) T_PersonList(novar X);
```

then the following type expression is *not* expanded:

$$\texttt{T\_List}(\alpha)$$

while the next one is:

$$(\alpha \preceq \texttt{T\_Person}).\texttt{T\_List}(\alpha)$$

The following is the formal definition of an expanded type.

Definition 4.8 (Expanded type). A simple constrained type $(\mathbb{C}).A$ (a constraint set $\mathbb{C}$) is called *expanded* if for every subexpression of the form $a(\vec{B})$ that occurs is $(\mathbb{C}).A$ (a constraint set $\mathbb{C}$) the following holds:
if

$$\textbf{type}\ (\mathbb{C}_1)\ a(\vec{\alpha});$$

is the type definition of a, then $\mathbb{C}_1[\vec{B}/\vec{\alpha}] \in \mathbb{C}$. □

The process that transforms a type into its expanded form is called *expansion* and is performed by recursively adding appropriate constraints to the constraint set. This process is described by the following algorithm.

Algorithm 4.1 (Type expansion).

Initialize Set S to be the set of all subexpressions in $\mathbb{C}$ and A which are neither variables nor 0-ary constructors; set R to be equal to $\mathbb{C}$; set S' to be the empty set

Add For each expression $a(\vec{B})$ in S, do the following:

1. Add $\mathbb{C}_1[\vec{B}/\vec{\alpha}]$ to R
2. Add all subexpressions in $\mathbb{C}_1[\vec{B}/\vec{\alpha}]$ which are neither variables nor 0-ary constructors to S'

where

$$\textbf{type}\ (\mathbb{C}_1)\ a(\vec{\alpha});$$

is the type definition of a

Iterate $S := S' \setminus S$; if the resulting set S is empty, succeed with the result $R.A$; otherwise, let $S' := \emptyset$ and continue from step **Add**

□

Expansion is guaranteed to terminate if there exists a valid ranking as will be shown in Chapter 5. For example, given the type definitions

$\textbf{type}\ (p \preceq \alpha)\ s(\alpha^-);$

$\textbf{type}\ (\alpha \preceq s(\alpha))\ q(\alpha^+);$

the following type expression is *not* expanded:

$$q(\beta)$$

However, it can be transformed into expanded form as follows:

$$q(\beta) \Rightarrow (\beta \preceq s(\beta)).q(\beta) \Rightarrow (\beta \preceq s(\beta), p \preceq \beta).q(\beta)$$

The expansion function $expand_{\mathcal{G}}$ provides the expanded form of a type expression. For example, $expand_{\mathcal{G}}(q(\beta)) = (\beta \preceq s(\beta), p \preceq \beta).q(\beta)$. Since the type expansion step is performed after the existence of a valid ranking for the user type graph has been established, the function $expand_{\mathcal{G}}$ is well-defined.

Convention. From now on, all types in the program are considered to be expanded. □

This can be done by means of the syntactic transformations:

1. Each type definition
   $\textbf{type}\ (\mathbb{C})\ a(\vec{\alpha});$
   is transformed into
   $\textbf{type}\ (\mathbb{C}')\ a(\vec{\alpha});$
   where $expand_{\mathcal{G}}((\mathbb{C}).a(\vec{\alpha})) = (\mathbb{C}').a(\vec{\alpha})$

2. Each subtype definition
   $(\mathbb{C})\ \underline{A} \preceq \overline{A};$
   is transformed into
   $(\mathbb{C}_{\underline{A}} \cup \mathbb{C}_{\overline{A}})\ \underline{A} \preceq \overline{A};$
   where $expand_{\mathcal{G}}((\mathbb{C}).\underline{A}) = (\mathbb{C}_{\underline{A}}).\underline{A}$ and $expand_{\mathcal{G}}((\mathbb{C}).\overline{A}) = (\mathbb{C}_{\overline{A}}).\overline{A}$
3. Each behavior definition
   behavior $(\mathbb{C})\ A \rightarrow R;$
   is transformed into
   behavior $(\mathbb{C}_A \cup \mathbb{C}_R)\ A \rightarrow R;$
   where $expand_{\mathcal{G}}((\mathbb{C}).A) = (\mathbb{C}_A).A$ and $expand_{\mathcal{G}}((\mathbb{C}).R) = (\mathbb{C}_R).R$
4. Each behavior association is transformed analogously to the behavior definition

Note that neither constant definitions

$$\texttt{const}\ A\ \langle \textbf{name} \rangle;$$

nor the *typed let* expression

$$\texttt{let}\ A\ \langle \textbf{name} \rangle = \ldots;$$

are transformed since A in both cases is not allowed to contain type variables; thus, the resulting constraints $\mathbb{C}$ are either inconsistent (which will be rejected by consistency checking) or, if consistent, $(\mathbb{C}).A = ().A$.

The expansion process is described here since it can only be done *after* the user type graph is successfully checked for simplicity (existence of valid ranking).

4.2.5 Constraint consistency

Constraint consistency checking is needed to identify erroneous *unsatisfiable constraints*, i.e. constraints that can not be satisfied by any possible type. This section provides the algorithm for constraint consistency checking in the given type system.

An intuitive reason for consistency checking is the fact that constrained types of the form $(\mathbb{C}).A$ are equivalent to the top type $\top$ in the hierarchy when their constraint sets $\mathbb{C}$ are unsatisfiable. Since it is the purpose of typechecking to ensure that data of the type $\top$ can never be generated by the program, the initial conditions described by type specifications given explicitely in the program are also required to specify types strictly less than $\top$. The last requirement is equivalent to the requirement of satisfyability of constraints of all types specified by the user.

Definition 4.9 (Constraint consistency). Each of the following constructs:

1. A type definition of the form
   type $(\mathbb{C})\ a(\vec{\alpha});$
2. A subtype definition of the form
   $(\mathbb{C})\ a(\vec{A}) \preceq b(\vec{B});$
3. A behavior definition of the form
   behavior $(\mathbb{C})\ A \rightarrow R$

has a *consistent constraint* iff

$$\vdash_{\mathcal{G}} \mathbb{C}$$

□

It is required that all type, subtype, and behavior definitions in the program have consistent constraints.

Note that this step can only be done *after* simplicity checking, since only in a sufficiently simple type system is the algorithm of Section 4.3 guaranteed to terminate.

Consider the following specification:

```
type T_Person;
type T_Child;
T_Child <= T_Person;
type (T_Child <= X) T_Array(novar X);
behavior (X <= T_Person) T_Array(X) -> X getOldest;
```

For brevity, p will be used for **T_Person**, c for **T_Child**, a for **T_Array**, and b for **getOldest**. The above definition can be represented in theoretical notation as

type p;

type c;

$c \preceq p$;

type $(c \preceq \alpha)\ s(\alpha^0)$;

behavior $(\alpha \preceq p)\ a(\alpha){\rightarrow}\alpha \quad b$;

Consider the question of constraint consistency for the behavior specification present in the above fragment. The expanded form is

type p;

type c;

$c \preceq p$;

type $(c \preceq \alpha)\ s(\alpha^0)$;

behavior $(\alpha \preceq p, c \preceq \alpha)\ a(\alpha){\rightarrow}\alpha \quad b$;

and the behavior constraint set is therefore

$$\mathbb{C} = \alpha \preceq p \wedge c \preceq \alpha$$

The formula

$$\vdash_{\mathcal{G}} \mathbb{C} \equiv \vdash_{\mathcal{G}} \alpha \preceq p \wedge c \preceq \alpha$$

is satisfied since there exists an α such that all constraints in $\mathbb{C}$ are satisfied (for example, $\alpha = c$). Therefore, the constraint is consistent.

On the other hand, if the subtype definition

```
T_Child <= T_Person;
```

was removed, the above constraint would be inconsistent. The reason for that is that now it is impossible to find an α to satisfy the formula

$$\alpha \preceq p \wedge c \preceq \alpha$$

since the latter requires that

$$c \preceq p$$

and this is no longer the case. Therefore, the formula

$$\vdash_{\mathcal{G}} \mathbb{C}$$

is not satisfied and the constraint is inconsistent.

In this section, the notion of *constraint consistency* was introduced. The procedure for checking constraint consistency has been established and exemplified.

4.2.6 Global behavior consistency

Behavior definitions have to be consistent. Since several behavior definitions can collectively define a single behavior, there are two aspects of consistency: local monotonicity (a single definition) and global monotonicity (a set of all definitions for the same behavior). The local monotonicity has been defined in Section 4.2.2. The global consistency related to interaction between different definitions of the same behavior is described in this section.

Note that local monotonicity conditions are not essential for the theory to be applicable. They are designed to reject the specifications that have a counterintuitive meaning, but in their absence typechecking algorithms will still be valid and subject reduction will be provable. On the other hand, the global monotonicity considered here is crucial. Without it, a correctly typechecked program can produce type errors during its execution.

Definition 4.10 (Global behavior consistency). A behavior b is *globally consistent* iff for every two definitions

$$\texttt{behavior}\ (\mathbb{C}_1)\ A_1 \rightarrow R_1\ \mathsf{b};$$

and

$$\texttt{behavior}\ (\mathbb{C}_2)\ A_2 \rightarrow R_2\ \mathsf{b};$$

the following holds:

$$(\mathbb{C}_1 \cup \mathbb{C}_2 \cup \{A_1 \preceq A_2\} \vdash_{\mathcal{G}} R_1 \preceq R_2)$$

□

All behaviors are required to be globally consistent. The algorithm for checking the entailment will be presented in Section 4.3.

Consider the following specifications:

```
type T_Real;
type T_Integer;
T_Integer <= T_Real;
behavior (T_Real, T_Real) -> T_Real add;     // 1
behavior (T_Integer, T_Integer) -> T_Integer add; // 2
```

For brevity, r will be used for **T_Real**, i for **T_Integer**, and b for **add**. The above definition can be represented in theoretical notation as

> **type** r;
>
> **type** i;
>
> $i \preceq r$;
>
> **behavior** $(r, r) \rightarrow r \quad b$; // 1
>
> **behavior** $(i, i) \rightarrow i \quad b$; // 2

In order to check the consistency, it is neccessary to check both directions: 1→2 and 2→1.

1. 1→2. Here, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $(r, r) \preceq (i, i)$ is unsatisfiable. Therefore, the consistency condition is trivially satisfied.
2. 2→1. Here, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $(i, i) \preceq (r, r)$ is trivially satisfied. It is left to show that

$$\vdash_{\mathcal{G}} i \preceq r$$

   which is immediately true. Therefore, the consistency condition in this case is also satisfied.

Thus, the above behavior specifications are globally consistent.

A more complicated example dealing with parametric types will be considered next.

```
type T_Set(covar X);
type T_Array(novar X);
T_Array(X) <= T_Set(X);
behavior T_Set(X) -> T_Set(X) someSubset;     // 1
behavior T_Array(X) -> T_Array(X) someSubset; // 2
```

For brevity, s will be used for **T_Set**, a for **T_Array**, and b for **someSubset**. The above definition can be represented in theoretical notation as

type $s(\alpha^+)$;

type $a(\alpha^0)$;

$a(\alpha) \preceq s(\alpha)$;

behavior $s(\alpha) \rightarrow s(\alpha) \quad b$; // 1

behavior $a(\alpha) \rightarrow a(\alpha) \quad b$; // 2

Again, it is necessary to check both directions:

1. 1→2. Here, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $s(\alpha) \preceq a(\beta)$ is unsatisfiable. Therefore, the consistency condition is trivially satisfied.
2. 2→1. Here, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $a(\alpha) \preceq s(\beta) \equiv \alpha \preceq \beta$ (since for any α and β, $\alpha \preceq \beta \equiv a(\alpha) \preceq s(\beta)$). It is left to show that

$$\alpha \preceq \beta \vdash_{\mathcal{G}} a(\alpha) \preceq s(\beta)$$

   which is immediately true. Therefore, the consistency condition in this case is also satisfied.

Thus, the above behavior specifications are globally consistent.

An example of an inconsistent behavior specification is given below.

```
type T_Set(covar X);
type T_MySet;
type T_Person;
T_MySet <= T_Set(X);
behavior T_Set(X) -> X pick;      // 1
behavior T_MySet -> T_Person pick; // 2
```

For brevity, s will be used for **T_Set**, m for **T_MySet**, p for **T_Person**, and b for **someSubset**. The above definition can be represented in theoretical notation as

type $s(\alpha^+)$;

type m;

type p;

$m \preceq s(\alpha)$;

behavior $s(\alpha) \rightarrow \alpha \quad b$; // 1

behavior $m \rightarrow p \quad b$; // 2

Checking both directions:

1. 1→2. Then, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $s(\alpha) \preceq m$ is unsatisfiable. Therefore, the consistency condition is trivially satisfied.

2. 2→1. Then, $\mathbb{C}_1 = \mathbb{C}_2 = \emptyset$, and $m \preceq s(\alpha) \equiv TRUE(\alpha)$ (since the constraint $m \preceq s(\alpha)$ trivially holds for all α). It is left to show that

$$TRUE(\alpha) \vdash_{\mathcal{G}} p \preceq \alpha$$

which is not true, since there exist α such that $p \preceq \alpha$ does not hold. Note that the presence of $TRUE(\alpha)$ on the left side of the turnstyle is crucial:

$$(TRUE(\alpha) \vdash_{\mathcal{G}} p \preceq \alpha) \equiv (\forall\alpha: \quad p \preceq \alpha) \equiv FALSE$$

while

$$(\vdash_{\mathcal{G}} p \preceq \alpha) \equiv (\exists\alpha: \quad p \preceq \alpha) \equiv TRUE$$

Therefore, the consistency condition in this case is not satisfied.

Thus, the above behavior specifications are not globally consistent and will be rejected. In order to see why these specifications are incorrect, consider an application of the behavior **pick** to an object of type **T_MySet** statically declared to be of type **T_Set(T_Student)**. Statically, the result type will be inferred to be **T_Student** according to the first specification. However, at run-time the call will be dispatched to the second association (as **T_MySet** is more specific than **T_Set**). However, the second association is only required to produce an object of type **T_Person**, and not necessarily of type **T_Student**. Thus, a run-time type error will occur.

In this section, the algorithm for checking global behavior consistency was described and exemplified. The next section deals with the issue of *functional consistency*, i.e. checking whether the specified function code corresponds to the type specifications given for that function.

4.2.7 Functional consistency

In this section, typechecking of function code will be described. All code in the target language (except for the "main" expression of the program) is contained in functions, and functions are associated with behaviors by *associations*. The algorithm presented in this section ensures that associations are locally consistent, i.e. that the function being associated indeed conforms to the type specified for it in the association.

Function consistency checking proceeds as follows. First, the *initial environment* Θ_0 associating constants in a program with their declared types is constructed. Then, for each function association its type is inferred and its validity is verified. Finally, the "main" expression of the program is typechecked. These verification steps are described below.

Definition 4.11 (Initial environment Θ_0). An *environment* Θ is defined as a set of name-to-type bindings of the form $\mathbf{v} : \mathrm{glb}_i(\mathbb{C}^i).A_i$. The *initial environment* Θ_0 for a program is defined as follows:

1. For each constant definition of the form

   const A **v**

   the association $\mathbf{v} : A$ is added to Θ_0

2. For each behavior **b**, the association $\mathbf{v} : \mathrm{glb}(X_1, \ldots, X_N)$ is added to Θ_0, where there is $X_i = (\mathbb{C}^i).A_i \rightarrow_b R_i$ for each behavior definition

   behavior $(\mathbb{C}^i)$ $A_i \rightarrow R_i$ **b**

   and for each behavior association

   association $(\mathbb{C}^i)$ $A_i \rightarrow R_i$ **b fun** ...

□

The denotation $\Theta(\mathbf{v})$ will be used to denote $\top$ if $\mathbf{v}$ is unbound in Θ and the type X if $\mathbf{v}$ is bound and its binding in Θ has the form $\mathbf{v} : X$. The operation $\uplus : \Theta_1 \uplus \Theta_2$ constructs the new typing environment that includes all bindings in Θ_2 and those bindings of Θ_1 that are not overridden in Θ_2. Formally,

$$(\Theta_1 \uplus \Theta_2)(\mathbf{v}) = \begin{cases} \Theta_2(\mathbf{v}) & \text{if } \Theta_2(\mathbf{v}) \neq \top \\ \Theta_1(\mathbf{v}) & \text{if } \Theta_2(\mathbf{v}) = \top \end{cases}$$

For example, for a set of definitions

```
const T_Integer MaxAge
behavior (T_Real, T_Real)→T_Real add
behavior (T_Integer, T_Integer)→T_Integer add
```

the initial environment will be

$$\Theta_0 = \{\mathbf{MaxAge} : \mathbf{T\_Integer},\ \mathbf{add} : \mathrm{glb}((\mathbf{T\_Real}, \mathbf{T\_Real}) \to_b \mathbf{T\_Real}, \\ (\mathbf{T\_Integer}, \mathbf{T\_Integer}) \to_b \mathbf{T\_Integer})\}$$

Definition 4.12 (Functional consistency). A program is *functionally consistent* iff

1. For each function association of the form

   $$\mathbf{association}\,(\mathbb{C})\ A \to R \quad \mathbf{b} \quad \mathbf{fun(x)}\ \langle\mathbf{expr}\rangle$$

   the following holds:

   $$\vdash_{\mathcal{G}} X \preceq (\mathbb{C})(A \to R)$$

   where $\Theta_0 \rhd$ **fun (x)** $\langle\mathbf{expr}\rangle : X = \mathrm{glb}_i((\mathbb{C}_i).A_i)$.

2. For the main expression $\langle\mathbf{expr}\rangle$ of the program, the following holds:

   $$\vdash_{\mathcal{G}} \mathbb{C}_i \text{ for at least one } i$$

   where $\Theta_0 \rhd \langle\mathbf{expr}\rangle : \mathrm{glb}_i((\mathbb{C}_i).A_i)$

□

Note that the condition for the functional expression can be equivalently written as

$$\exists i: \quad \mathbb{C} \vdash_{\mathcal{G}} \mathbb{C}_i \wedge (A_i \preceq A \to R)$$

The inference algorithm computing the relationship $\rhd$ is presented in Figure 4.2. The intuitive meaning of the statement of the form

$$\Theta \rhd \mathbf{expr} : X$$

is that the expression **expr** has type X when all constants are typed according to Θ. The denotation $\mathrm{glb}_{j_1,\ldots,j_n} E(j_1,\ldots,j_n)$ (where $E(j_1,\ldots,j_n)$ is some type expression depending on $j_1,\ldots,j_n$) is defined as

$$\mathrm{glb}_{j_1,\ldots,j_n} E_{j_1,\ldots,j_n} \overset{\mathrm{def}}{=} \mathrm{glb}\{E_{j_1,\ldots,j_n} \mid (j_1,\ldots,j_n) \in (\mathrm{dom}(j_1) \times \cdots \times \mathrm{dom}(j_n))\} \tag{4.3}$$

$$\frac{\Theta(\mathrm{u}) = \mathrm{glb}_i((\mathbb{C}^i).A^i) \qquad \Theta(\mathrm{u}) \neq \top}{\Theta \rhd \mathbf{u} : \mathrm{glb}_i((\mathbb{C}^i).A^i)}\ \mathsf{Axiom}$$

$$\frac{\Theta \uplus \{\mathbf{x} : \alpha\} \rhd \langle\mathbf{expr}\rangle : \mathrm{glb}_i((\mathbb{C}^i).A^i)}{\Theta \rhd \mathbf{fun}\ (\mathbf{x})\ \langle\mathbf{expr}\rangle : \mathrm{glb}_i((\mathbb{C}^i).\alpha \to A^i)}\ \mathsf{Abs}$$
where α is a fresh type variable

$$\frac{\Theta \rhd \langle\mathbf{expr}\rangle_1 : \mathrm{glb}_i((\mathbb{C}_1^i).A_1^i) \qquad \Theta \rhd \langle\mathbf{expr}\rangle_2 : \mathrm{glb}_j((\mathbb{C}_2^j).A_2^j)}{\Theta \rhd \langle\mathbf{expr}\rangle_1(\langle\mathbf{expr}\rangle_2) : \mathrm{glb}_{i,j}(\mathbb{C}_1^i \wedge \mathbb{C}_2^j \wedge \{A_1^i \preceq A_2^j \to \alpha\}).\alpha}\ \mathsf{Appl}$$
where α is a fresh type variable

$$\frac{\Theta \rhd \langle\mathbf{expr}\rangle_1 : \mathrm{glb}_{j_1}((\mathbb{C}_1^{j_1}).A_1^{j_1}) \qquad \dots \qquad \Theta \rhd \langle\mathbf{expr}\rangle_n : \mathrm{glb}_{j_n}((\mathbb{C}_n^{j_n}).A_n^{j_n})}{\Theta \rhd (\langle\mathbf{expr}\rangle_1, \dots, \langle\mathbf{expr}\rangle_n) : \mathrm{glb}_{j_1,\dots,j_n}((\wedge_i \mathbb{C}_i^{j_i}).(A_1^{j_1}, \dots, A_n^{j_n}))}\ \mathsf{Product}$$

$$\frac{\Theta \rhd \langle\mathbf{exprs}\rangle : \mathrm{glb}_i((\mathbb{C}_1^i).A_1^i) \qquad \Theta \rhd \langle\mathbf{expr}\rangle : \mathrm{glb}_j((\mathbb{C}_2^j).A_2^j)}{\Theta \rhd \langle\mathbf{exprs}\rangle; \langle\mathbf{expr}\rangle : \mathrm{glb}_{i,j}((\mathbb{C}_1^i \wedge \mathbb{C}_2^j).A_2^j)}\ \mathsf{Seq}$$

$$\frac{\Theta \rhd \langle\mathbf{expr}\rangle_1 : \mathrm{glb}_i((\mathbb{C}_1^i).A_1^i) \qquad \Theta \uplus \{\mathbf{x} : \mathrm{glb}_i((\mathbb{C}_1^i).A_1^i)\} \rhd \langle\mathbf{expr}\rangle_2 : \mathrm{glb}_j((\mathbb{C}_2^j).A_2^j)}{\Theta \rhd \mathbf{let}\ \mathbf{x} = \langle\mathbf{expr}\rangle_1\ \mathbf{in}\ \langle\mathbf{expr}\rangle_2 : \mathrm{glb}_{i,j}((\mathbb{C}_1^i \wedge \mathbb{C}_2^j).A_2^j)}\ \mathsf{Let}$$

$$\frac{\Theta \rhd \langle\mathbf{expr}\rangle_1 : \mathrm{glb}_i((\mathbb{C}_1^i).A_1^i) \qquad \Theta \uplus \{\mathbf{x} : T\} \rhd \langle\mathbf{expr}\rangle_2 : \mathrm{glb}_j((\mathbb{C}_2^j).A_2^j)}{\Theta \rhd \mathbf{let}\ T\ \mathbf{x} = \langle\mathbf{expr}\rangle_1\ \mathbf{in}\ \langle\mathbf{expr}\rangle_2 : \mathrm{glb}_{i,j}((\mathbb{C}_1^i \wedge \mathbb{C}_2^j \wedge \{A_1^i \preceq T\}).A_2^j)}\ \mathsf{TypedLet}$$

Figure 4.2: Typing rules for the target language

Example 4.1. Consider the following set of specifications:

```
const T_Integer MaxAge
behavior (T_Real,T_Real)→T_Real add
behavior (T_Integer,T_Integer)→T_Integer add
association (T_Integer)→T_Integer addMaxAge
    fun(x) {return add(x,MaxAge);}
```

The initial environment is

$$\Theta_0 = \{\mathbf{MaxAge} : \mathbf{T\_Integer}, \\ \mathbf{add} : \mathrm{glb}((\mathbf{T\_Real}, \mathbf{T\_Real}) \to_b \mathbf{T\_Real}, \\ (\mathbf{T\_Integer}, \mathbf{T\_Integer}) \to_b \mathbf{T\_Integer}), \\ \mathbf{addMaxAge} : \mathrm{glb}((\mathbf{T\_Integer} \to_b \mathbf{T\_Integer}))\}$$

In order to typecheck the body of the behavior **addMaxAge**, it is necessary to find X such that

$$\Theta_0 \rhd \mathbf{fun(x)\ add(x,\ MaxAge)} : X \tag{4.4}$$

Let

$$X \equiv \mathrm{glb}_i((\mathbb{C}_0^i).A_0^i) \tag{4.5}$$

Using the rules in Figure 4.2, the following derivation is constructed.
Using the rule Abs, (4.4) is true if

$$\mathrm{glb}_i((\mathbb{C}_0^i).A_0^i) \equiv \mathrm{glb}_i(\mathbb{C}_1^i).\alpha \to A_1^i \tag{4.6}$$

and

$$\Theta_0 \uplus \{\mathbf{x} : \alpha\} \rhd \mathbf{add(x,\ MaxAge)} : X_0$$

where

$$X_0 \equiv \mathsf{glb}_i((\mathbb{C}_1^i).A_1^i) \tag{4.7}$$

Using the rule Appl, the above is true if

$$\mathsf{glb}_i((\mathbb{C}_1^i).A_1^i) \equiv \mathsf{glb}_{i,j}(\mathbb{C}_2^{i,1} \wedge \mathbb{C}_2^{j,2} \wedge \{A_2^{i,1} \preceq A_2^{j,2} \rightarrow \beta\}).\beta \tag{4.8}$$

and (branch 1)

$$\Theta_0 \uplus \{\mathtt{x} : \alpha\} \rhd \mathbf{add} : \mathsf{glb}_i((\mathbb{C}_2^{i,1}).A_2^{i,1})$$

and (branch 2)

$$\Theta_0 \uplus \{\mathtt{x} : \alpha\} \rhd (\mathtt{x}, \mathtt{MaxAge}) : \mathsf{glb}_j((\mathbb{C}_2^{j,2}).A_2^{j,2})$$

Following branch 1: using rule Axiom (since **add** is in Θ_0), the following is proven:

$$\mathsf{glb}_i((\mathbb{C}_2^{i,1}).A_2^{i,1}) \equiv \mathsf{glb}((\mathtt{T\_Real}, \mathtt{T\_Real}) \rightarrow_b \mathtt{T\_Real}, (\mathtt{T\_Integer}, \mathtt{T\_Integer}) \rightarrow_b \mathtt{T\_Integer}) \tag{4.9}$$

Following branch 2: using rule Product:

$$\mathsf{glb}_j((\mathbb{C}_2^{j,2}).A_2^{j,2}) \equiv \mathsf{glb}_{j_1,j_2}((\mathbb{C}_3^{j_1,1} \wedge \mathbb{C}_3^{j_2,2}).(A_3^{j_1,1}, A_3^{j_2,2})) \tag{4.10}$$

where (branch 2.1)

$$\Theta_0 \uplus \{\mathtt{x} : \alpha\} \rhd \mathtt{x} : \mathsf{glb}_{j_1}((\mathbb{C}_3^{j_1,1}).A_3^{j_1,1})$$

and (branch 2.2)

$$\Theta_0 \uplus \{\mathtt{x} : \alpha\} \rhd \mathtt{MaxAge} : \mathsf{glb}_{j_2}((\mathbb{C}_3^{j_2,2}).A_3^{j_2,2})$$

Following branch 2.1: using the rule Axiom (since `x` is in the set of assumptions), it is proven that

$$\mathsf{glb}_{j_1}((\mathbb{C}_3^{j_1,1}).A_3^{j_1,1}) \equiv \alpha \tag{4.11}$$

Following branch 2.2: using the rule Axiom (since `MaxAge` is in Θ_0), it is proven that

$$\mathsf{glb}_{j_2}((\mathbb{C}_3^{j_2,2}).A_3^{j_2,2}) \equiv \mathtt{T\_Integer} \tag{4.12}$$

All branches have successfully terminated. Now it is possible to synthesize the resulting type expression X by following the derivation tree bottom-up:
from (4.11):

$$\mathsf{glb}_{j_1}((\mathbb{C}_3^{j_1,1}).A_3^{j_1,1}) \equiv \alpha \tag{4.13}$$

from (4.12):

$$\mathsf{glb}_{j_2}((\mathbb{C}_3^{j_2,2}).A_3^{j_2,2}) \equiv \mathtt{T\_Integer} \tag{4.14}$$

from (4.13), (4.14), and (4.10):

$$\mathsf{glb}_j((\mathbb{C}_2^{j,2}).A_2^{j,2}) \equiv (\alpha, \mathtt{T\_Integer}) \tag{4.15}$$

from (4.9):

$$\begin{aligned}\mathsf{glb}_i((\mathbb{C}_2^{i,1}).A_2^{i,1}) \equiv \mathsf{glb}(&(\mathtt{T\_Real}, \mathtt{T\_Real}) \rightarrow_b \mathtt{T\_Real},\\ &(\mathtt{T\_Integer}, \mathtt{T\_Integer}) \rightarrow_b \mathtt{T\_Integer})\end{aligned} \tag{4.16}$$

from (4.15), (4.16), and (4.8):

$$\begin{aligned}\mathrm{glb}_i((\mathbb{C}_1^i).A_1^i) \equiv \mathrm{glb}(&((\mathbf{T\_Real},\mathbf{T\_Real}){\rightarrow_b}\mathbf{T\_Real} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).\beta,\\ &((\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow_b}\mathbf{T\_Integer} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).\beta)\end{aligned} \qquad (4.17)$$

From (4.7) and (4.17):

$$\begin{aligned}X_0 \equiv \mathrm{glb}(&((\mathbf{T\_Real},\mathbf{T\_Real}){\rightarrow_b}\mathbf{T\_Real} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).\beta,\\ &((\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow_b}\mathbf{T\_Integer} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).\beta)\end{aligned} \qquad (4.18)$$

The type X_0 is the type inferred for the function body.
Finally, from (4.18), (4.7), (4.6), and (4.5)

$$\begin{aligned}X \equiv \mathrm{glb}(&((\mathbf{T\_Real},\mathbf{T\_Real}){\rightarrow_b}\mathbf{T\_Real} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).(\alpha{\rightarrow}\beta),\\ &((\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow_b}\mathbf{T\_Integer} \preceq (\alpha,\mathbf{T\_Integer}){\rightarrow}\beta).(\alpha{\rightarrow}\beta))\end{aligned} \qquad (4.19)$$

According to Definition 4.12, one of the following conditions should be satisfied for the specification under consideration to be functionally correct:

$$\begin{aligned}&\vdash_{\mathcal{G}} ((\mathbf{T\_Real},\mathbf{T\_Real}){\rightarrow_b}\mathbf{T\_Real} \preceq (\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow}\beta)\\ &\qquad\qquad \wedge (\alpha{\rightarrow}\beta \preceq \mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer})\end{aligned} \qquad (4.20)$$

or

$$\begin{aligned}&\vdash_{\mathcal{G}} ((\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow_b}\mathbf{T\_Integer} \preceq (\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow}\beta)\\ &\qquad\qquad \wedge (\alpha{\rightarrow}\beta \preceq \mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer})\end{aligned} \qquad (4.21)$$

where the premises are empty since

$$\mathit{expand}_{\mathcal{G}}(\mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer}) \equiv \mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer}$$

Formula (4.20) is false since

$$\begin{aligned}&\vdash_{\mathcal{G}}((\mathbf{T\_Real},\mathbf{T\_Real}){\rightarrow_b}\mathbf{T\_Real} \preceq (\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow}\beta)\\ &\qquad \wedge (\alpha{\rightarrow}\beta \preceq \mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} ((\mathbf{T\_Integer},\mathbf{T\_Integer}) \preceq (\mathbf{T\_Real},\mathbf{T\_Real}))\\ &\qquad \wedge (\mathbf{T\_Real} \preceq \beta) \wedge (\mathbf{T\_Integer} \preceq \alpha) \wedge (\beta \preceq \mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} (\mathbf{T\_Integer} \preceq \mathbf{T\_Real}) \wedge (\mathbf{T\_Real} \preceq \beta) \wedge (\mathbf{T\_Integer} \preceq \alpha) \wedge (\beta \preceq \mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} (\mathbf{T\_Real} \preceq \beta) \wedge (\beta \preceq \mathbf{T\_Integer}) \wedge (\mathbf{T\_Integer} \preceq \alpha)\\ &\equiv \mathit{FALSE}\end{aligned}$$

since $\mathbf{T\_Real} \not\preceq \mathbf{T\_Integer}$.
However, formula 4.21 is true since

$$\begin{aligned}&\vdash_{\mathcal{G}}((\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow_b}\mathbf{T\_Integer} \preceq (\mathbf{T\_Integer},\mathbf{T\_Integer}){\rightarrow}\beta)\\ &\qquad \wedge (\alpha{\rightarrow}\beta \preceq \mathbf{T\_Integer}{\rightarrow}\mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} ((\mathbf{T\_Integer},\mathbf{T\_Integer}) \preceq (\mathbf{T\_Integer},\mathbf{T\_Integer}))\\ &\qquad \wedge (\mathbf{T\_Integer} \preceq \beta) \wedge (\mathbf{T\_Integer} \preceq \alpha) \wedge (\beta \preceq \mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} (\mathbf{T\_Integer} \preceq \mathbf{T\_Integer}) \wedge (\mathbf{T\_Integer} \preceq \beta) \wedge (\mathbf{T\_Integer} \preceq \alpha) \wedge (\beta \preceq \mathbf{T\_Integer})\\ &\equiv \vdash_{\mathcal{G}} (\beta = \mathbf{T\_Integer}) \wedge (\mathbf{T\_Integer} \preceq \alpha)\\ &\equiv \mathit{TRUE}\end{aligned}$$

and therefore there is α and β such that the above constraint is satisfied.
Therefore, the definition of **addMaxAge** is considered to be functionally consistent. □

In this section, the notion of *functional consistency* was introduced and the algorithm for its checking has been described. The type inferencing rules used in functional consistency checking were defined. The type inferencing rules presented in this section together with the entailment verification algorithm given in Section 4.3 constitute the core of the type and consistency checking.

4.2.8 Dispatch consistency

A uniform behavioral system in which every action is performed via a behavior application has to support an efficient run-time dispatch mechanism. In theory, a dispatch mechanism can be similar in expressibility to the type specification mechanism of a language. However, the expressiveness of a dispatch mechanism is inversely proportional to its complexity and, therefore, to its efficiency. Considering the fact that the dispatch mechanism is the main run-time engine of a behavioral system, a certain compromise between expressiveness and efficiency is necessary.

The material in this section is related to a particular dispatch mechanism that has been chosen for the described system. This mechanism supports multiple dispatch, but ignores differences between different parametric types from the same parametric type family. For example, it is possible to dispatch the calls **equal(aPoint,aPoint)** and **equal(aPoint,aColorPoint)** to different functions. However, the dispatch of the calls **aStudentSet.pick** and **aPersonSet.pick** is *required* to yield the same code.

Note that the *validity* of the calls *can* depend on a particular type parameter (e.g the call **aNumberList.sort** is type-correct while **anObjectList.sort** is not). Once the call is valid, however, its dispatch should only depend on the primary functors of the types involved.

There are several aspects related to the dispatch consistency. First, for every concrete type and any type-correct behavior application involving that type, the code to dispatch to should be defined (no "message not understood" errors on statically type-correct calls). Second, the code to execute should be *unambiguous* (no "message ambiguous" errors). Finally, the dispatch should always yield a function that was type-checked with respect to types of actual arguments (or their supertypes).

The last issue is the consequence of the compromise being made between the expressive power of dispatch mechanism and its efficiency. Namely, if the expressiveness of the dispatch mechanism was equivalent to the expressiveness of the rest of the type system, this issue would not arise.

The following material describes the algorithms used to check all three aspects of dispatch consistency.

Since the dispatch occurs at a coarser level (concrete, primary-functor only types) than a type definition does, for dispatch consistency checking it is necessary to be able to deal with the behavior associations at both levels. While the type specification is given directly in a behavior association, the *concrete form* of the type has to be inferred. The following algorithms are used to derive that form from the type specification given.

Definition 4.13 (Concrete type constructors). A type constructor c in $\mathcal{G}$ is called *concrete* iff its corresponding type definition is of the form

$$\textbf{concrete type } (\mathbb{C})\ c(\ldots);$$

The set of all concrete type constructors in a given user type graph $\mathcal{G}$ is denoted $Concrete_{\mathcal{G}}$. □

Definition 4.14 (Constructor products). A tuple $\langle c_1, \ldots, c_n\rangle$ is called *a constructor product over* $\mathcal{G}$ if each of c_i is a type constructor in $\mathcal{G}$. The relationship $\leq_{\mathcal{G}}$ defined by the graph $\mathcal{G}$ on type constructors is extended to constructor products in the following way:

$$\langle \underline{c}_1, \ldots, \underline{c}_n\rangle \leq_{\mathcal{G}} \langle \bar{c}_1, \ldots, \bar{c}_m\rangle \overset{\text{def}}{=} (n = m \wedge \bigwedge_i \underline{c}_i \leq_{\mathcal{G}} \bar{c}_i)$$

The set of all constructor products of arity n is denoted $Constr_{\mathcal{G}}{}^n$. □

Definition 4.15 (Primary form). A *primary form* of a simple constrained type $X = (\mathbb{C}).(A_1,\dots,A_N)$ is a constructor product (denoted $primary_{\mathcal{G}}(X)$) defined as follows:

$$primary_{\mathcal{G}}(X) \overset{\text{def}}{=} \left\{ \langle c_1,\dots,c_n\rangle \mid c_i = \begin{cases} pfunctor(A_i) & \text{if } A_i \neq \alpha \\ \mathrm{glb}_{\mathcal{G}}\{pfunctor(u_i^j) \mid \{\alpha \preceq u_i^j\} \in \mathbb{C}\}_j & \text{otherwise if } A_i = \alpha \end{cases} \right\}$$

□

Here the usual convention for unary products ($\langle A\rangle \equiv A$) is used.

The primary form is basically a simplification of a given type. The given type is simplified up to a point when it can be presented as a constructor product. This simplified form is the one used for dispatch. As an example, consider the type

$$X = (\alpha \preceq \mathtt{T\_Integer} \wedge \alpha \preceq \mathtt{T\_Real}).(\alpha, \mathtt{T\_List}(\alpha)) \tag{4.22}$$

Its primary form is $\langle\mathtt{T\_Integer}, \mathtt{T\_List}\rangle$. Indeed, for the first component we must compute the greatest lower bound of **T_Integer** and **T_Real**, which is **T_Integer**. For the second, the primary functor **T_List** is taken according to the definition above.

Definition 4.16 (Abstract form). An abstract form of a constructor product $t = \langle t_1,\dots,t_n\rangle$ (denoted $abstract_{\mathcal{G}}(t)$) is defined as follows:

$$abstract_{\mathcal{G}}(t) \overset{\text{def}}{=} expand_{\mathcal{G}}((t_1(\alpha_1^1,\dots,\alpha_1^{Arity(t_1)}),\dots,t_n(\alpha_n^1,\dots,\alpha_n^{Arity(t_n)})))$$

where α_j^i are fresh type variables. □

Computing an abstract form of a constructor product is, in a sense, an inverse to taking a primary form of a type. The latter transforms a constrained type into a constructor product, while the former transforms a constructor product into a constrained type. However, since some information is lost when the primary form is obtained, abstract form of a primary form of a given constrained type is in general different from the original. For example, the abstract from of a constructor product $\langle\mathtt{T\_Integer}, \mathtt{T\_List}\rangle$ is

$$(\mathtt{T\_Integer}, \mathtt{T\_List}(\alpha))$$

which is not the same as X (4.22).

Definition 4.17 (Concrete set). A *concrete set* for a set of simple constrained types $\{X_i = (\mathbb{C}_i).A_i\}_{i=1}^n$ (denoted $concrete_{\mathcal{G}}(\{X_i\}_{i=1}^n)$) is defined as follows:
If $A_i = (A_i^1,\dots,A_i^m)$ for all i, then

$$\begin{aligned} concrete_{\mathcal{G}}(X) \overset{\text{def}}{=} \{c \mid\ & c \in Concrete_{\mathcal{G}}{}^m \\ & \wedge c \leq_{\mathcal{G}} primary_{\mathcal{G}}(X_1) \wedge \cdots \wedge c \leq_{\mathcal{G}} primary_{\mathcal{G}}(X_n) \\ & \wedge \nexists c' \in concrete_{\mathcal{G}}(X)\colon\ c' \neq c \wedge c \leq_{\mathcal{G}} c' \\ & \wedge \left(\bigwedge_i \mathbb{C}_i \wedge \mathbb{C}_c \vdash_{\mathcal{G}} \mathbb{C}_{c'} \wedge A_c \preceq A_{c'} \wedge \bigwedge_i (A_{c'} \preceq A_i)\right)\} \end{aligned}$$

otherwise

$$concrete_{\mathcal{G}}(X) \overset{\text{def}}{=} \emptyset$$

where $(\mathbb{C}_c).A_c = abstract_{\mathcal{G}}(c)$ and $(\mathbb{C}_{c'}).A_{c'} = abstract_{\mathcal{G}}(c')$ □

Informally, a concrete set is a set of concrete constructor products c that are the "maximal" ones below all of the types X_i. Maximality here is understood in the following sense: c dominates c' in the concrete set if it is above c', and its abstract form (which is a type X_c) has the property $X'_c \preceq X_c \preceq \text{glb}_i X_i$, where X'_c is the abstract form of c'.

Consider the following example. Let the following be the specifications given in the program

```
type T_Ordered(contravar X);
type T_Numeric subtype of T_Ordered(T_Numeric);
concrete type T_Char subtype of T_Ordered(T_Char);
concrete type T_Real subtype of T_Numeric;
concrete type T_Integer subtype of T_Real;
```

and the constrained types X_1 and X_2 given by

$$X_1 = (\alpha \preceq \mathtt{T\_Ordered}(\alpha)).(\alpha, \alpha)$$
$$X_2 = (\mathtt{T\_Integer}, \mathtt{T\_Ordered}(\alpha))$$

The primary form of X_1 is

$$\langle \mathtt{T\_Ordered}, \mathtt{T\_Ordered} \rangle$$

while the primary form of X_2 is

$$\langle \mathtt{T\_Integer}, \mathtt{T\_Ordered} \rangle$$

The concrete set for X_1 is

$$\{\langle \mathtt{T\_Real}, \mathtt{T\_Real} \rangle, \langle \mathtt{T\_Real}, \mathtt{T\_Char} \rangle, \langle \mathtt{T\_Char}, \mathtt{T\_Real} \rangle, \langle \mathtt{T\_Char}, \mathtt{T\_Char} \rangle\}$$

and for X_2 (as well for (X_1, X_2))

$$\{\langle \mathtt{T\_Integer}, \mathtt{T\_Real} \rangle, \langle \mathtt{T\_Integer}, \mathtt{T\_Char} \rangle\}$$

Another example that shows how the last condition in Definition 4.17 works is given below.

```
type T_A1;
type T_A2;
type T_B;
type (X <= T_A1) T_C1(covar X);
T_C1(X) <= T_B;
type (X <= T_A2) T_C2(covar X);
T_C2(X) <= T_C1(X);
```

The concrete set for the type T_B will be {T_C1, T_C2}. In order to see why T_C2 is in this set even though it is below T_C1, we write the elimination condition from Definition 4.17 (here c = T_C2 and c' = T_C1):

$$\alpha \preceq \mathtt{T\_A2} \vdash_G \beta \preceq \mathtt{T\_A1} \wedge \mathtt{T\_C2}(\alpha) \preceq \mathtt{T\_C1}(\beta) \wedge \mathtt{T\_C1}(\beta) \preceq \mathtt{T\_B}$$

This is not true (e.g., take α = T_A2 and the left-hand side becomes unsatisfiable as it implies T_A2 $\preceq$ T_A1 which is not the case). Therefore, T_C2 can not be eliminated from the concrete set for T_B even though T_C2 is below T_C1 in the graph. Informally, it is not eliminated because there are types of the form T_C2(α) that are subtypes of T_B, but not subtypes of any of the types T_C1(β).

The behavior coverage condition ensures that for every behavior definition and for every concrete product that satisfies its conditions there is at least one association for this behavior that has its conditions satisfied by the same concrete product. Informally, it means that every behavior definition is covered by at least one behavior association.

Definition 4.18 (Behavior coverage). A behavior b *satisfies coverage condition* iff

$$\forall i \in [1,\dots,N_d]\ \forall t \in concrete_{\mathcal{G}}((\mathbb{C}_i^d).A_i^d)\ \exists j \in [1,\dots,N_a]:$$
$$\mathbb{C}_t \wedge \mathbb{C}_i^d \wedge \{A_t \preceq A_i^d\} \vdash_{\mathcal{G}} \{A_t \preceq A_j^a \preceq A_i^d\} \wedge \mathbb{C}_j^a$$

where $(\mathbb{C}_t).A_t = abstract_{\mathcal{G}}(t)$,

$$\texttt{behavior}\ (\mathbb{C}_i^d)\ A_i^d \rightarrow R_i^d \qquad (i \in [1,\dots,N_d])$$

are the behavior definitions for b, and

$$\texttt{association}\ (\mathbb{C}_i^a)\ A_i^a \rightarrow R_i^a\ \dots \qquad (i \in [1,\dots,N_a])$$

are the behavior associations for b. □

Consider the following example. Let the following be the specifications given in the program:

```
type T_Ordered(contravar X);
type T_Numeric subtype of T_Ordered(T_Numeric);
concrete type T_Char subtype of T_Ordered(T_Char);
concrete type T_Real subtype of T_Numeric;
concrete type T_Integer subtype of T_Real;
behavior (T_Ordered(X), T_Ordered(X)) -> T_Boolean compare;
association (T_Char, T_Char) -> T_Boolean compare implementation fun1;
association (T_Real, T_Real) -> T_Boolean compare implementation fun2;
```

Then, according to the definition, we have to find an appropriate association for each type in the concrete set of the type $X_1 = (\texttt{T\_Ordered}(\alpha), \texttt{T\_Ordered}(\alpha))$ (using α for the free type variable **X**). The concrete set is

$$\{\langle\texttt{T\_Real},\texttt{T\_Real}\rangle, \langle\texttt{T\_Real},\texttt{T\_Char}\rangle, \langle\texttt{T\_Char},\texttt{T\_Real}\rangle, \langle\texttt{T\_Char},\texttt{T\_Char}\rangle\}$$

For the constructor product $\langle\texttt{T\_Real},\texttt{T\_Real}\rangle$ and the second association the condition is

$$\begin{aligned}&(\texttt{T\_Real},\texttt{T\_Real}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \vdash_{\mathcal{G}}\\ &\qquad (\texttt{T\_Real},\texttt{T\_Real}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha))\end{aligned}$$

and is immediately satisfied. For the constructor product $\langle\texttt{T\_Char},\texttt{T\_Char}\rangle$ and the first association the condition is

$$\begin{aligned}&(\texttt{T\_Char},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \vdash_{\mathcal{G}}\\ &\qquad (\texttt{T\_Char},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha))\end{aligned}$$

For the constructor product $\langle\texttt{T\_Real},\texttt{T\_Char}\rangle$ and the first association the condition is

$$\begin{aligned}&(\texttt{T\_Real},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \vdash_{\mathcal{G}}\\ &\qquad (\texttt{T\_Char},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha)) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha))\end{aligned}$$

which is satisfied since

$$\begin{aligned}&(\texttt{T\_Real},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha))\\ &\qquad \Rightarrow \alpha \preceq \texttt{T\_Numeric} \wedge \alpha \preceq \texttt{T\_Char}\\ &\qquad \Rightarrow (\texttt{T\_Char},\texttt{T\_Char}) \preceq (\texttt{T\_Ordered}(\alpha),\texttt{T\_Ordered}(\alpha))\end{aligned}$$

The conditions for the constructor product $\langle\texttt{T\_Char},\texttt{T\_Real}\rangle$ are satisfied in the same way.

Thus, there is at least one association covering each constructor product in the concrete set and therefore the coverage condition for the above specification is satisfied. Note that we were able to confirm coverage even though there is no association for ⟨T_Numeric, T_Numeric⟩ since this constructor product is abstract and therefore does not need to be covered.

The next condition guarantees that there is only one most specific association for each concrete type. This in turn guarantees the absence of "message ambiguous" errors at run-time. This condition is standard for the systems with multiple dispatch and is formulated in terms of constructor products, not types.

Definition 4.19 (Behavior unambiguity). A behavior b *satisfies unambiguity condition* iff

$$\begin{aligned}
&\forall i,j \in [1,\dots,N_a]:\\
&\quad primary_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \vee primary_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)\\
&\quad \vee\ \forall t \in concrete_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a, (\mathbb{C}_j^a).A_j^a):\\
&\qquad\quad \exists k \in [1,\dots,N_a]:\ \ t \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a)\\
&\qquad\qquad \wedge\ primary_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)\\
&\qquad\qquad \wedge\ primary_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a)
\end{aligned}$$

where $(\mathbb{C}_t).A_t = abstract_{\mathcal{G}}(t)$ and

$$\textbf{association}\ (\mathbb{C}_i^a)\ A_i^a \rightarrow R_i^a\ \dots \qquad (i \in [1,\dots,N_a])$$

are the behavior associations for b. □

The *correctness* condition is designed to check whether all type-correct behavior applications of a particular behavior are dispatched to a function that was type-checked in accordance with the actual argument types. This condition is necessary because dispatch occurs at a coarser granularity than does type specification. All behaviors in the system are required to satisfy this condition.

Definition 4.20 (Behavior correctness). A behavior b *satisfies correctness condition* iff

$$\begin{aligned}
\forall i,j \in [1,\dots,N_a]:\ \ & primary_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \Rightarrow\\
&((\forall t \in concrete_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a, abstract_{\mathcal{G}}(primary_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a))):\\
&\quad \mathbb{C}_t \wedge \mathbb{C}_j^a \wedge \{A_t \preceq A_j^a\} \vdash_{\mathcal{G}} \mathbb{C}_i^a \wedge \{A_t \preceq A_i^a \preceq A_j^a\})
\end{aligned}$$

where $(\mathbb{C}_t).A_t = abstract_{\mathcal{G}}(t)$ and

$$\textbf{association}\ (\mathbb{C}_i^a)\ A_i^a \rightarrow R_i^a\ \dots \qquad (i \in [1,\dots,N_a])$$

are the behavior associations for b. □

Consider the following set of specifications:

```
concrete type T_Set(covar X);
concrete type T_SpecialSet(covar X) subtype of T_Set(X);
association T_Set(X) -> T_Boolean isEmpty fun1;
association (X <= T_Person) T_SpecialSet(X) -> T_Boolean isEmpty fun2;
```

The second function is only supposed to be invoked for special sets of persons (or their subtypes). However, since dispatch uses only primary functors of argument types, it is unable to distinguish between **T_SpecialSet(T_Person)** and **T_SpecialSet(T_Object)** and dispatches to the second function (as the more specific one) in both cases. This causes the body of the second function to be executed under more liberal assumptions then the ones used when its body was typechecked and can therefore cause run-time type errors. Therefore the above set of specifications is disallowed. Indeed, the behavior correctness condition in this case is (for the type constructor T_SpecialSet which is in the concrete set of (T_Set,T_SpecialSet)):

$$\texttt{T\_SpecialSet}(\alpha) \preceq \texttt{T\_Set}(\alpha) \vdash_{\mathcal{G}} \\ \alpha \preceq \texttt{T\_Person} \wedge \texttt{T\_SpecialSet}(\alpha) \preceq \texttt{T\_SpecialSet}(\alpha) \preceq \texttt{T\_Set}(\alpha)$$

which is equivalent to

$$TRUE(\alpha) \vdash_{\mathcal{G}} \alpha \preceq \texttt{T\_Person}$$

and is false as there exists an α such that $\alpha \preceq$ T_Person does not hold.

Definition 4.21 (Dispatch). The function $dispatch_{\mathcal{G}}(b, q) : \mathcal{B} \times RTyp_{\mathcal{G}} \to [0, \ldots, N_a]$ is defined as follows:

$$dispatch_{\mathcal{G}}(b,q) \overset{\text{def}}{=} \begin{cases} i: & \exists s: \ \langle s, i\rangle \in S^b_{\min}(t) \quad \text{if } |S^b_{\min}(t)| = 1 \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{B}$ is the set of all behaviors,

$$t \overset{\text{def}}{=} primary_{\mathcal{G}}(q)$$
$$S^b(t) \overset{\text{def}}{=} \{\langle s, i\rangle \mid s = primary_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i) \wedge t \leq_{\mathcal{G}} s\}$$
$$S^b_{\min}(t) \overset{\text{def}}{=} \{\langle s, i\rangle \mid \langle s, i\rangle \in S^b(t) \wedge \nexists \langle s', i'\rangle \in S^b(t): \ \ s' \leq_{\mathcal{G}} s\}$$

and

$$\textbf{association } (\mathbb{C}^a_i)\ A^a_i \to R^a_i \ \ldots \qquad (i \in [1, \ldots, N_a])$$

are the behavior associations for b. □

In this definition, the notion of *run-time type* (Definition 5.20, domain $RTyp_{\mathcal{G}}$) has been used along with the appropriate extension of $primary_{\mathcal{G}}(\cdot)$ given in Definition 5.3.3. Intuitively, a run-time type is a type of an object that is present during program execution. Every run-time type is also a type, but the opposite is not true. For example, an abstract type is not a run-time type because an object of such a type can never be created and thus can not be present at run-time. Some other restictions also apply.

The dispatch process chooses the most specific association with respect to the type that is obtained by taking the argument type and cutting off its leaves. For example, the type T_Collection(T_Person) becomes T_Collection, while the product type (T_Collection(T_Person),T_Integer) becomes ⟨T_Collection, T_Integer⟩. If the most specific association does not exist (either there are no suitable associations, or there are several suitable associations that are incomparable to each other), then the dispatch fails and returns 0. This can never happen in a successfully typechecked program, as will be proven in Chapter 5.

In this section, the notion of dispatch consistency was introduced. The algorithms testing various aspects of dispatch consistency were presented and illustrated by examples.

The next section introduces the core algorithm of the presented type system.

4.3 Entailment

The algorithms and notions presented in this section constitute the core of the presented consistency checking algorithms. The algorithms of this section always deal with *expanded* types and constraints.

The entailment relationship $\bigvee_i \mathbb{C}^1_i \vdash \bigvee_j \mathbb{C}^2_j$ can be understood as a logical formula

$$\forall \vec{\alpha}(\bigvee_i \mathbb{C}^1_i \Rightarrow \exists \vec{\beta}: \ \bigvee_j \mathbb{C}^2_j)$$

where

$$\vec{\alpha} = \cup_i FTV(\mathbb{C}^1_i) \setminus \cup_j FTV(\mathbb{C}^2_j)$$

and $\vec{\beta} = \cup_j FTV(\mathbb{C}^2_j)$. The task of the main entailment algorithm is to determine whether a logical formula of this form is satisfied in a type system defined by the given user type graph $\mathcal{G}$.

Algorithm 4.2 (Deciding entailment). Let $\bigvee_i \mathbb{C}^1_i \vdash_{\mathcal{G}} \bigvee_j \mathbb{C}^2_j$ be the entailment that is necessary to check. The algorithm operates as follows:

1. For each i, check $\mathbb{C}^1_i \vdash_{\mathcal{G}} \bigvee_j \mathbb{C}^2_j$ in the following manner:
   (a) Check whether $\vdash_{\mathcal{G}} \mathbb{C}^1_i$ as follows: attempt to flatten $\mathbb{C}^1_i$ w.r.t $\mathcal{G}$. If it fails, the formula $\vdash_{\mathcal{G}} \mathbb{C}^1_i$ can not be proven and the algorithm ends with success (a contradiction implies everything); otherwise, continue to the next step
   (b) Change all variables $\alpha_k \in FTV(\mathbb{C}^1_i)$ into 0-ary type constructors a_k. Form the extended type graph $\mathcal{G}^i_e$ by adding edges corresponding to constraints from $\mathbb{C}^1_i$ and vertices corresponding to type constructors a_k to $\mathcal{G}$.
   (c) If $\mathcal{G}^i_e$ does not have a valid ranking, end the algorithm with the result *UNKNOWN*.
   (d) For each j, attempt to flatten $\mathbb{C}^2_j[\vec{a}/\vec{\alpha}]$ w.r.t $\mathcal{G}^i_e$. If at least one of them succeeds, succeed; otherwise, fail.
2. If all checks above succeed, succeed; otherwise, fail

□

For example, in order to check the entailment

$$\alpha \preceq \beta, \beta \preceq \gamma \vdash \alpha \preceq \gamma$$

the entailment $\vdash \alpha \preceq \beta, \beta \preceq \gamma$ is checked first. Since the formula on the right of the turnstyle is already flattened, the check succeeds and the algorithm proceeds to the next step. Here, the "types" a, b, and g are introduced into the type graph $\mathcal{G}_e$ with the following relationships between them:

$$a \preceq b, b \preceq g$$

$\mathcal{G}_e$ thus obtained has a valid ranking, and the algorithm now attempts to prove

$$\vdash_{\mathcal{G}_e} a \preceq g$$

by flattening the formula $a \preceq g$ w.r.t. $\mathcal{G}_e$. The flattening succeeds, and therefore the algorithm finishes with success.

Algorithm 4.3 (Flatten). Let $\mathbb{C}$ is the constraint formula to be flattened.

Initialization

1. Create an initially empty set D of already checked constraints
2. Create an initially empty set E of eliminated variables
3. For each variable α_i, create initially empty sets of lower and upper bounds (L_i and U_i).
4. Create a directed graph G with vertices α_i and initially no edges
5. Create a list of complex constraints C. Set C to $\mathbb{C}$ (arbitrarily ordered).

Process For each constraint $A \preceq B$ in C, do:

1. Remove the constraint from C

2. Clear the set of new constraints N
3. If $A = B$ or $A \preceq B \in D$, continue to the next constraint
4. Otherwise, add the constraint to D and do the following:
   (a) If $A = \alpha_i$, $B = \alpha_j$, and there is no path from α_i to α_j in G, then do the following:
      i. Add the edge $\alpha_i \rightarrow \alpha_j$ to G.
      ii. If the addition of the edge creates cycles in G, replace all variables $\alpha_{i_1}, \ldots, \alpha_{i_n}$ involved in the cycles in all expressions used in the algorithm by the variable α_{i_1}. Collapse vertices $\alpha_{i_1}, \ldots, \alpha_{i_n}$ into the single vertex α_{i_1} in G. New sets U'_{i_1} and L'_{i_1} for this vertex are constucted as follows: $U'_{i_1} := \cup_j U_{i_j}$ and $L'_{i_1} := \cup_j U_{i_j}$. For each pair $\langle l, u\rangle \in (L'_{i_1} \times U'_{i_1})$, add the constraint $l \preceq u$ to N. Add the set $\{\alpha_{i_1} = \alpha_{i_k}\}_{k=2}^{n}$ to E.
      iii. Otherwise (no cycles) form sets

$$S_l = \cup_k L_k, \text{where } k \in \{k \mid \text{there is a path from } \alpha_k \text{ to } \alpha_i \text{ in } G\}$$
$$S_u = \cup_m U_m, \text{where } m \in \{m \mid \text{there is a path from } \alpha_j \text{ to } \alpha_m \text{ in } G\}$$

      For each pair $\langle l, u\rangle \in S_l \times S_u$, add the constraint $l \preceq u$ to N.
   (b) Otherwise, if $A = \text{lub}_i(A_i)$, add constraints of the form $A_i \preceq B$ to N.
   (c) Otherwise, if $B = \text{glb}_i(B_i)$, add constraints of the form $A \preceq B_i$ to N
   (d) Otherwise, if $A = \alpha_i$, add B to the set U_i. Form the set

$$S_l = \cup_k L_k, \text{where } k \in \{k \mid \text{there is a path from } \alpha_k \text{ to } \alpha_i \text{ in } G\}$$

   Add constraints of the form $l \preceq B$ ($l \in S_l$) to N.
   (e) Otherwise, if $B = \alpha_i$, add A to the set L_i. Form the set

$$S_u = \cup_k U_k, \text{where } k \in \{k \mid \text{there is a path from } \alpha_i \text{ to } \alpha_k \text{ in } G\}$$

   Add constraints of the form $A \preceq u$ ($u \in U_k$) to N.
   (f) Otherwise, if $A = a(\vec{A})$ and $B = a(\vec{B})$, add the constraints $A_i \preceq^{Var(a,i)} B_i$ to N. Here $Var(a, i) \in \{+, -, 0\}$ is the defined variance of the i-th parameter of type constructor a, $\preceq^{+} \equiv \preceq$, $\preceq^{-} \equiv \succeq$, $\preceq^{0} \equiv =$, and addition of a constraint $A = B$ is treated as addition of two constraints $A \preceq B$ and $B \preceq A$.
   (g) Add constraints in $N \setminus (C \cup D)$ to the tail of C
   (h) If the current constraint was not processed on this step, remove it from D and put it back into the head of C (so that the algorithm does not try to process it now).

Resolution Empty the set of sets T. For each constraint $A \preceq B$ in C, do:

1. Remove the constraint from C and add it to D
2. If $A = \text{glb}_i(A_i)$, add the set $\{A_i \preceq B\}$ to T.
3. Otherwise, if $B = \text{lub}_i(B_i)$, add the set $\{A \preceq B_i\}$ to T.
4. Otherwise, $A = a(\vec{A_i})$ and $B = b(\vec{B_i})$. Do the following:
   (a) If there is no path from a to b in $\mathcal{G}$, fail
   (b) Otherwise, for each path $P = \langle a = c_0, c_1, \ldots, c_n = b\rangle$ from a to b in $\mathcal{G}$, add C' to set

T. Here C' is the set of the following constraints:

$$\begin{aligned}
A_i &\preceq^{Var(c_0,i)} C_i^{\dot{0}\to 1};\\
&\mathbb{C}^0[\vec{C}^{\dot{0}\to 1}]\\
&\mathbb{C}^{0\to 1}[\vec{C}^{0\to \dot{1}}];\\
&\mathbb{C}^1[\vec{C}^{0\to \dot{1}}];\\
C_i^{0\to \dot{1}} &\preceq^{Var(c_1,i)} C^{\dot{1}\to 2};\\
&\mathbb{C}^1[\vec{C}^{\dot{1}\to 2}];\\
&\mathbb{C}^{1\to 2}[\vec{C}^{1\to \dot{2}}];\\
&\mathbb{C}^2[\vec{C}^{1\to \dot{2}}];\\
C_i^{1\to \dot{2}} &\preceq^{Var(c_2,i)} C^{\dot{2}\to 3};\\
&\vdots\\
C_i^{n-1\to \dot{n}} &\preceq B_i
\end{aligned}$$

where

$$(\mathbb{C}^{k-1\to k})\ c_{k-1}(\vec{C}^{\widehat{k-1}\to k}) \preceq c_k(\vec{C}^{k-1\to \dot{k}})$$

are the labels of the edges in P and

$$\textbf{type}\ (\mathbb{C}^k)\ c_k(\vec{V}_{c_k,i})$$

are the labels of vertices in P. All new variables introduced on this step are considered to be fresh.

Subgoals Create a cross-product of all sets in T: $T' = \times_{t\in T} t$. Empty the result set R. For each $t' \in T'$, do:

1. Clone the state of the algorithm ($G' = G \cup FTV(t')$, $D' = D$, $U_i' = U_i$, $L_i' = L_i$, $C' = C \cup t'$, $E' = \emptyset$). The lower and upper bounds sets for the variables in $FTV(t') \setminus vertices(G)$ are set to empty sets.
2. Solve the resulting task starting from the step **Process**.
3. If there was a success, add the result to R

If none of the tasks above succeeds, fail.

Finalization If C is empty, succeed; the flattened form is

$$(\bigvee_{r\in R} r) \vee (\bigwedge_i (\bigwedge_{l\in L_i} l \preceq \alpha_i \wedge \bigwedge_{u\in U_i} \alpha_i \preceq u) \wedge \bigwedge_{(\alpha_i\to\alpha_j)\in G} \alpha_i \preceq \alpha_j) \wedge \bigwedge_{e\in E} e)$$

Otherwise, continue from step **Process**.

□

Example 4.2. Consider the following specification:

```
type T_Set(covar X);
type T_List(novar X);
T_List(X) <= T_Set(X);
```

and the formula $\texttt{T\_List}(\alpha) \preceq \texttt{T\_Set}(\beta)$.

Initialization $D := \emptyset$, $L_\alpha = L_\beta = U_\alpha = U_\beta = \emptyset$, $vertices(G) = \{\alpha, \beta\}$, $C := \{\mathtt{T\_List}(\alpha) \preceq \mathtt{T\_Set}(\beta)\}$

Process Skip as the first and only condition in C does not have the form suitable for processing at this step

Resolution $T := \emptyset$. Since the constraint has the form $\mathtt{T\_List}(\vec{A}) \preceq \mathtt{T\_Set}(\vec{B})$ and there is a path from $\mathtt{T\_List}$ to $\mathtt{T\_Set}$ in $\mathcal{G}$, the following conditions form the set C':

$$C' = \{\alpha \preceq^0 \gamma, \gamma \preceq^+ \beta\} = \{\alpha \preceq \gamma, \gamma \preceq \alpha, \gamma \preceq \beta\}$$

where γ is a fresh variable. Now, $T := T \cup C' = C'$,

$$C := C \setminus \{\mathtt{T\_List}(\alpha) \preceq \mathtt{T\_Set}(\beta)\} = \emptyset$$

and

$$D := D \cup \{\mathtt{T\_List}(\alpha) \preceq \mathtt{T\_Set}(\beta)\} = \{\mathtt{T\_List}(\alpha) \preceq \mathtt{T\_Set}(\beta)\}$$

Subgoals $R := \emptyset$; the new subtask is formed:

$$G' = G \cup \gamma, \quad D' = D, \quad U_i' = U_i (i \in \{\beta, \alpha\}), \quad L_i' = L_i (i \in \{\beta, \alpha\}),$$
$$L_\gamma' = U_\gamma' = \emptyset, C' = C \cup T = \{\alpha \preceq \gamma, \gamma \preceq \alpha, \gamma \preceq \beta\}$$

Subgoal 1

Process Processing the first constraint $c = \alpha \preceq \gamma$: adding edge $\alpha \rightarrow \gamma$ to G, removing c from C, inserting c in D.

Process Processing the second constraint $c = \gamma \preceq \alpha$: an attempt to add the edge $\gamma \rightarrow \alpha$ to G leads to a cycle. Collapsing α and γ to α. Removing c from C, inserting c in D, adding $\{\alpha = \gamma\}$ to E.

Process Processing the second constraint $c = \alpha \preceq \beta$ (note that due to the change made in the previous step, the variable on the left was changed from γ to α): adding the edge $\alpha \rightarrow \beta$ to G.

Finalization (Skipping to finalization since C is empty): The result is

$$\alpha \preceq \beta \wedge \alpha = \gamma$$

The subgoal is achieved - returning one level back

There was a success. $R := R \cup \{\alpha \preceq \beta \ \wedge \alpha = \gamma\} = \{\alpha \preceq \beta \wedge \alpha = \gamma\}$.

Finalization Set C is empty; succeed with the result

$$\alpha \preceq \beta \wedge \alpha = \gamma$$

Thus, the flattened form of $\mathtt{T\_List}(\alpha) \preceq \mathtt{T\_Set}(\beta)$ in the given type system was found to be $F = \alpha \preceq \beta \wedge \alpha = \gamma$. Since γ is a fresh variable which is not used anywhere else, F is equivalent to $\alpha \preceq \beta$. □

In this section, the core (entailment/flattening) algorithm was presented. A simple example was used to illustrate the algorithm.

4.4 Conclusions

In this chapter, the typechecking algorithm for the proposed type system has been presented. The algorithm works in several stages. First, the source language program is translated to a simplified target language. Second, the target language program is analyzed and typechecked. During typechecking, several aspects of program consistency are verified. For aspects related to dispatch, a separate algorithm is used. The algorithm described in Section 4.2.8 is a generalization of algorithms developed in [ADL91, Ghe91, MHH91, BSG95, Cas95b, CGL95, CL95, CL97, QKB96, Sha97].

The core of the rest of the typechecking system is the entailment/flattening algorithm presented in Section 4.3, which is a generalization of algorithms presented in [BM96b] and [Pot96].

In the next chapter, the formal aspects of the type system presented so far will be discussed.

Chapter 5

The Type System Theory

The set of rules and algorithms described in the previous chapter defines a procedure for typechecking a program in the target language. As with any type system, two main questions arise:

1. Does the typechecking algorithm always terminate? (decidability)
2. If a program is successfully typechecked, does it guarantee the absence of type errors at run-time? (correctness)

In this chapter, both of these questions will be answered. It will be proven that the type system presented in this dissertation is both decidable and correct. Decidability of the algorithms will be discussed in Section 5.1.

Section 5.2 presents a theoretical framework for reasoning about types. It will be proven that the core entailment algorithms give correct results with respect to this framework.

In Section 5.3, the natural semantics of the target language is presented. Using the framework established in the previous section, the subject reduction theorem for the given semantics of the target language will be proven. This will establish correctness of the typechecking algorithms presented in the previous chapter.

Section 5.4 describes introduction of imperative types in the type framework established so far, gives semantics to the objects of these types, and proves that the correctness of the typechecking is not affected by the introduction of such types. This section serves a dual purpose. First, it demonstrates how the type-theoretical framework established so far can be extended to deal with types not present in the basic type set of the framework. Second, it shows that the addition of imperative types to the target language does not affect any aspects of the typechecking, including algorithms and the correctness results.

Section 5.5 discusses various possible extensions to the natural semantics and its handling by the type system. While the primary purpose of this chapter is to develop and present the type system theory rather than an object-oriented language semantics, it is important to address the typing issues that might arise during the development of the full-fledged version of such semantics.

In Section 5.6, various features of the presented theory are discussed and the main results of this chapter are summarized.

5.1 Decidability

In this section, decidability of Algorithms 4.1 and 4.3 will be proven. All the other algorithms in Chapter 4 are trivially terminating as they are either non-recursive or their recursion is based on the structure of a given finite expression.

The decidability proofs in this section are based on Lemma 5.1 that will be proven in the following section.

5.1.1 Finite evolution

Definition 5.1 (Evolution system). The *evolution system* $\mathcal{E}$ is a tuple $\langle E, R, I, r : E \to R, S_0\rangle$, where E is a set (elements of this set will be called *e-atoms*), R is a totally ordered set of *ranks*, r is a ranking function that assigns a rank from R to each element of E ($r(e) = r_e$), I is a finite set of indices, and S_0 is the *initial state* (S_0 is a finite subset of E, $S_0 \subseteq E$).

The *evolution process* is a process of transforming the state of the evolution system:

$$\langle S_0, S_1, S_2, \ldots\rangle$$

It consists of *evolution steps* which transform the current state of the system S_k into its next state S_{k+1}.

Each evolution step consists of zero or more reproductions and exactly one survival, where reproductions and survival are defined below:

Individual Reproduction A single e-atom (*parent*) e is chosen from S_k and a single index i is chosen from I. Their pairing $\langle e, i\rangle$ results in one or more *children* $\{c_i\}_{i=1}^{N}$ such that

$$\max_i\{r(c_i)\} < r(e)$$

The set of children is added to S_{k+1}.

Paired Reproduction Two e-atoms (*parents*) e_1, e_2 are chosen from S_k. Their combination results in one or more *children* $\{c_i\}_{i=1}^{N}$ such that

$$\max_i\{r(c_i)\} < \min\{r(e_1), r(e_2)\}$$

The set of children is added to S_{k+1}.

Survival A subset of *surviving e-atoms* S' is chosen from S_k ($S' \subseteq S_k$) and added to S_{k+1}.

In addition, each evolution process has the following property:
During an evolution process each pair

$$\langle o_1, o_2\rangle \in (E \times E) \cup (E \otimes I)$$

can participate in reproduction no more than once. Here

$$A \otimes B \overset{\text{def}}{=} (A \times B) \cup (B \times A)$$

□

An evolution system can be thought of as a system of living creatures that evolve in discrete time. These creatures possess some genetic property (rank) that diminishes after each reproduction.

Since each evolution step must decrease the ranking, there can be a state in which the system can no longer evolve. If all strictly decreasing sequences in R are finite, it is intuitively obvious that any evolution system will eventually (in a finite number of steps) come to the point when the evolution stops. This intuitive observation is supported by Lemma 5.1. The following definition gives precise meaning to the notion of a *downward-finite set* which is intuitively equivalent to an ordered set with no infinitely decreasing sequences.

Definition 5.2 (Downward-finite set). A set E is *downward-finite* if

1. E is totally ordered
2. There are no decreasing infinite sequences in E, i.e. there are no sets e_i ($i = 1, \ldots, \infty$, $e_i \in E$) such that the following holds: $i < j \Rightarrow e_i > e_j$.

□

For example, the set of natural numbers is downward-finite, while the set of all real numbers between 0 and 1 is not.

Lemma 5.1 (Finite evolution). If $\mathcal{E} = \langle E, R, I, r : E \to R, S_0 \rangle$ is an evolution system and the rank set R is downward-finite, then

1. the number of evolution steps involving reproduction is finite,
2. the union of all states of any given evolution process contains a finite number of e-atoms
3. if $\mathcal{E}$ is such that there can be no infinite sequence of survival-only steps, then there is no infinite evolution process for $\mathcal{E}$, i.e. each evolution process contains a finite number of steps.

□

Proof. The second statement of the lemma follows from the first one, the finiteness of the initial state, and the fact that each evolution step adds no more than a finite number of e-atoms.

The third statement of the lemma also trivially follows from the first one.

Thus, it is sufficient to prove the first statement. The proof will proceed by constructing a series of finite sets U_k and T_k such that

$$S_k \subseteq U_k \tag{5.1}$$
$$U_{k+1} = U_k \cup T_k \tag{5.2}$$
$$T_k = \emptyset \Rightarrow T_{k+1} = \emptyset \tag{5.3}$$
$$T_{k+1} = \emptyset \ \vee \ \max\{r(e) \mid e \in T_{k+1}\} < \max\{r(e) \mid e \in T_k\} \tag{5.4}$$

Once such sets are constructed, the following reasoning is used. Let

$$r_k = \max\{r(e) \mid e \in T_k\}$$

Then, $r_k > r_{k+1}$ and (since R is downward-finite)

$$\exists N: \quad \forall i \geq N: \quad T_i = \emptyset$$

and therefore

$$\forall i \geq N: \quad U_i = U_N$$

thus

$$\forall i: \quad S_i \subseteq U_N$$

from which

$$\forall i: \quad ((S_i \times S_i) \cup (S_i \otimes I)) \subseteq ((U_N \times U_N) \cup (U_N \otimes I)) = \hat{U}$$

$\hat{U}$ is finite since U_N and I are finite. Each evolution step "uses up" at least one pair from $\hat{U}$. Therefore, there can be no more than K reproduction steps, where K is the cardinality of $\hat{U}$. Thus, the total number of reproductive evolution steps is finite.

In order to complete the proof it is sufficient to show that the series of sets T_k and U_k satisfying the conditions (5.1)–(5.4) can be constructed for any given evolution process.

Assume $P = \langle S_0, S_1, \ldots \rangle$ is an evolution process for the evolution system $\mathcal{E}$. Construct sets T_i and U_i in the following way:

1. $U_0 = S_0$

2. $U_{k+1} = U_k \cup T_{k+1}$

3. $T_0 = S_0$

4. $T_{k+1} = \bigcup_{(o_1,o_2)\in(T_k\otimes U_k)\cup(T_k\otimes I)} C(o_1,o_2)$

where $C(o_1,o_2)$ is the set of children that was produced when the pair $\langle o_1,o_2\rangle$ was used in reproduction during the evolution process P. If the pair was never used, $C(o_1,o_2) = \emptyset$. $C(o_1,o_2)$ is well-defined, since each pair can be used no more than once during a given evolution process according to the definition of the evolution system.

Now it will be shown that the sets constructed above satisfy the conditions (5.1)–(5.4).

Condition (5.1) First, an auxiliary statement will be proven. It will be shown that

$$\bigcup_{(o_1,o_2)\in(U_k\times U_k)\cup(U_k\otimes I)} C(o_1,o_2) \subseteq U_{k+1} \tag{5.5}$$

Proof by induction.

Base ($k=0$):

$$\begin{aligned}
&\bigcup_{(o_1,o_2)\in(U_0\times U_0)\cup(U_0\otimes I)} C(o_1,o_2)\\
&= \bigcup_{(o_1,o_2)\in(S_0\times S_0)\cup(S_0\otimes I)} C(o_1,o_2)\\
&= T_1 \subseteq (U_0 \cup T_1) = U_1
\end{aligned}$$

Induction Assumption:

$$\bigcup_{(o_1,o_2)\in(U_k\times U_k)\cup(U_k\otimes I)} C(o_1,o_2) \subseteq U_{k+1}$$

Then,

$$\begin{aligned}
&\bigcup_{(o_1,o_2)\in(U_{k+1}\times U_{k+1})\cup(U_{k+1}\otimes I)} C(o_1,o_2)\\
&= \bigcup_{(o_1,o_2)\in(U_k\times U_k)\cup(U_k\otimes T_{k+1})\cup(T_{k+1}\times T_{k+1})\cup(U_k\otimes I)\cup(T_{k+1}\otimes I)} C(o_1,o_2)\\
&= \bigcup_{(o_1,o_2)\in(U_k\times U_k)\cup(U_k\otimes I)} C(o_1,o_2)\\
&\quad \cup \bigcup_{(o_1,o_2)\in(U_k\otimes T_{k+1})\cup(T_{k+1}\times T_{k+1})\cup(U_k\otimes I)\cup(T_{k+1}\otimes I)} C(o_1,o_2)\\
&\subseteq U_{k+1} \cup \bigcup_{(o_1,o_2)\in(U_k\otimes T_{k+1})\cup(T_{k+1}\times T_{k+1})\cup(U_k\otimes I)\cup(T_{k+1}\otimes I)} C(o_1,o_2)\\
&= U_{k+1} \cup \bigcup_{(o_1,o_2)\in((U_k\cup T_{k+1})\otimes T_{k+1})\cup(T_{k+1}\otimes I)} C(o_1,o_2)\\
&= U_{k+1} \cup T_{k+2} = U_{k+2}
\end{aligned}$$

Now it is possible to show that the condition (5.2) is satisfied. The proof is again by induction:

Base ($k=0$): $S_0 \subseteq S_0 = U_0$

Induction Assumption: $S_k \subseteq U_k$. Let P_{k+1} be the set of parent pairs of reproductions at the $(k+1)$'st step, and $C_{k+1}(o_1, o_2)$ be the set of children produced by the pair $\langle o_1, o_2 \rangle$ at the $(k+1)$'st step. Then, $C_{k+1}(o_1, o_2) = C(o_1, o_2)$ and

$$P_{k+1} \subseteq (S_k \times S_k) \cup (S_k \otimes I)$$

Thus

$$\begin{aligned}
S_{k+1} &= (S_k \cup \bigcup_{\langle o_1, o_2 \rangle \in P_{k+1}} C_{k+1}(o_1, o_2)) \setminus S'_k \\
&= (S_k \cup \bigcup_{\langle o_1, o_2 \rangle \in P_{k+1}} C(o_1, o_2)) \setminus S'_k \\
&\subseteq S_k \cup \bigcup_{\langle o_1, o_2 \rangle \in P_{k+1}} C(o_1, o_2) \\
&\subseteq U_k \cup \bigcup_{\langle o_1, o_2 \rangle \in P_{k+1}} C(o_1, o_2) \\
&\subseteq U_k \cup \bigcup_{\langle o_1, o_2 \rangle \in (S_k \times S_k) \cup (S_k \otimes I)} C(o_1, o_2) \\
&\subseteq U_k \cup \bigcup_{\langle o_1, o_2 \rangle \in (U_k \times U_k) \cup (U_k \otimes I)} C(o_1, o_2) \\
&\subseteq U_k \cup U_{k+1} = U_{k+1}
\end{aligned}$$

Condition (5.2) Immediate from definition of U_k

Condition (5.3) Immediate from definition of T_k

Condition (5.4) By definition of reproduction,

$$\max\{r(c) \mid c \in C(o_1, o_2)\} < \min\{r(o_1), r(o_2)\}$$

Therefore (provided $T_k \neq \emptyset$ and $T_{k+1} \neq \emptyset$),

$$\begin{aligned}
&\max\{r(o) \mid o \in T_{k+1}\} \\
&\quad = \max\{r(o) \mid o \in \bigcup_{\langle o_1, o_2 \rangle \in (T_k \otimes U_k) \cup (T_k \otimes I)} C(o_1, o_2)\} < \max\{r(o) \mid o \in T_k\}
\end{aligned}$$

Thus, it has been shown that a sequence of sets U_k and T_k satisfying the conditions (5.1)–(5.4) can be constructed for any evolution process. Therefore, the lemma has been proven. □

5.1.2 Termination of expansion algorithm

Algorithm 4.1 is used to find an expanded form for a given simple constrained type $(\mathbb{C}).A$ w.r.t the given user type graph $\mathcal{G}$. The following theorem establishes the termination of this algorithm for any constrained type provided $\mathcal{G}$ has valid ranking (see Definition 4.7).

Theorem 5.1 (Termination of expansion algorithm). Algorithm 4.1 terminates for any constrained type $(\mathbb{C}).A$ provided $\mathcal{G}$ has a valid ranking. □

Proof. Let $\bar{r}(\cdot)$ be a valid ranking in $\mathcal{G}$. Define ranking $r(\cdot)$ for each type A that is not a type variable or a 0-ary constructor as $r(\cdot) = \bar{r}(\mathit{pfunctor}(A))$. The rank set will be a set of natural numbers less or equal to the maximum rank in $\bar{r}(\cdot)$. The set S used in Algorithm 4.1 is an evolution system and each iteration of the algorithm is its reproduction step. Therefore, according to Lemma 5.1, the algorithm terminates for any simple constrained type $(\mathbb{C}).A$. □

5.1.3 Termination of flattening algorithm

Algorithm 4.3 is used to find a flattened form for a given constraint set $\mathbb{C}$ w.r.t the given user type graph $\mathcal{G}$. The following theorem establishes the termination of this algorithm for any finite constraint set provided $\mathcal{G}$ has valid ranking (see Definition 4.7).

Theorem 5.2 (Termination of flattening algorithm). Algorithm 4.3 terminates for any finite constraint set ($\mathbb{C}$) provided $\mathcal{G}$ has a valid ranking. $\square$

Proof. First, the finiteness of the step **Process** will be demonstrated. The step **Process** is a cycle that adds new constraints to the list of constraints C it iterates over; it only passes through newly added constraints that were added to the tail of the list. It also modifies the graph G and the sets U_i and L_i of upper and lower bounds for each participating variable. For each constraint that is added to the tail of the list, the algorithm checks whether this constraint has already been processed (processed constraints are stored in set D). Consider the set W of all type expressions participating in constraints in $C \cup (\cup_i U_i) \cup (\cup_i L_i)$. Assume the initial set C is finite. Then, W is finite as well and therefore there is only a finite number of possible constraints $w_l \preceq w_u$ such that w_l and w_u are subexpressions of some type expressions from W. Such constraints constitute a (finite) set X. The algorithm for the step **Process** never creates new expressions; it only decomposes the type expressions in W to produce new constraints. Thus,

1. One constraint is eliminated from the remainder of C on every iteration;
2. Every constraint in C is processed no more then once (due to the usage of the set of already processed constraints D);
3. Constraints that are added to the remainder (tail) of C are taken from the finite set X;
4. The set X is the cycle invariant.

Therefore, the iteration must stop no later than on its x-th step, where x is the cardinality of X. Thus, the finiteness of the step **Process** is proven.

Next, the finiteness of the whole algorithm will be shown. First, a ranking of type expressions will be introduced. This ranking will later be used to define an evolution system related to the algorithm. Lemma 5.1 will then be applied to the resolution system to yield the algorithm termination.

Let $\bar{r}(\cdot)$ be a valid ranking in $\mathcal{G}$. Let $N(i)$ be the number of type constructors c in $\mathcal{G}$ such that $\bar{r}(c) = i$. To each type constructor c, assign a number $M(c)$ in the range $[1, N(\bar{r}(c))]$ in such a way that

$$c \neq d \wedge \bar{r}(c) = \bar{r}(d) \Rightarrow M(c) \neq M(d)$$

Define a new ranking

$$\bar{r}'(a) \overset{\text{def}}{=} \begin{cases} 2 + M(a) + \sum_{j=1}^{\bar{r}(a) - \min_{c \in \mathcal{G}} \bar{r}(c) + 1} j \times N(j) & a \text{ is a type constructor} \\ 1 & a = \text{glb} \\ 2 & a = \text{lub} \end{cases}$$

This new ranking has two properties:

1. It is a valid ranking since it does not change relative ordering of constructor ranks, i.e.

   $$\bar{r}(a) < \bar{r}(b) \Rightarrow \bar{r}'(a) < \bar{r}'(b)$$

2. It assigns a unique number to each non-0-ary type constructor in $\mathcal{G}$ as well as to type operators lub and glb.

Let $\bar{r}'_{max}$ be the maximum rank in this new ranking. Define a *simplified tree representation* of a type as its tree representation stripped off 0-ary type constructors, and define ranking $r(\cdot)$ for each type A as a tuple $\langle n_{\bar{r}'_{max}}, \dots, n_1\rangle$, where n_i is the maximum height of the vertices corresponding to the type constructor a_i with the rank $\bar{r}'(a_i) = i$ in the simplified tree representation of type A or zero if the constructor a_i does not participate in A. For example, if $\bar{r}'(a) = 3$, $\bar{r}'(b) = 4$, and $\bar{r}'(c) = 5$, then $\bar{r}(a(\alpha, a(b(b(\beta)), \alpha))) = \langle 0, 3, 5, 0, 0\rangle$, $\bar{r}(a(c(\alpha))) = \langle 2, 0, 3, 0, 0\rangle$, and $\bar{r}(\alpha) = \langle 0, 0, 0, 0, 0\rangle$. Let the tuples be lexicographically ordered. Then the set of such tuples comprises the ranking set R which is downward-finite.

Let I be the (finite) set of 0-ary type constructors in $\mathcal{G}$. Let the *connector set* of type expressions $A = a(\vec{A})$ and $B = b(\vec{B})$ along the path $P = \langle a = c_0, c_1, \dots, c_n = b\rangle$ in $\mathcal{G}$ be the set $N(A, B, P)$ defined as follows:

$$
\begin{aligned}
\{A_i\} \setminus I &\subseteq N(A, B, P)\\
\{C_i^{\dot{0}\to 1}\} \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^0[\vec{C}^{\dot{0}\to 1}]) \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^{0\to 1}[\vec{C}^{0\to \dot{1}}]) \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^1[\vec{C}^{0\to \dot{1}}]) \setminus I &\subseteq N(A, B, P)\\
\{C_i^{0\to \dot{1}}\} \setminus I &\subseteq N(A, B, P)\\
\{C^{\dot{1}\to 2}\} \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^1[\vec{C}^{\dot{1}\to 2}]) \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^{1\to 2}[\vec{C}^{1\to \dot{2}}]) \setminus I &\subseteq N(A, B, P)\\
(\text{all type expressions in } \mathbb{C}^2[\vec{C}^{1\to \dot{2}}]) \setminus I &\subseteq N(A, B, P)\\
\{C_i^{1\to \dot{2}}\} \setminus I &\subseteq N(A, B, P)\\
\{C^{\dot{2}\to 3}\} \setminus I &\subseteq N(A, B, P)\\
&\vdots\\
\{C_i^{n-1\to \dot{n}}\} \setminus I &\subseteq N(A, B, P)\\
\{B_i\} \setminus I &\subseteq N(A, B, P)
\end{aligned}
$$

where

$$(\mathbb{C}^{k-1\to k})\ c_{k-1}(\vec{C}^{\widehat{\dot{k-1}}\to k}) \preceq c_k(\vec{C}^{k-1\to \dot{k}})$$

are the labels of the edges in P and

$$\mathbf{type}\ (\mathbb{C}^k)\ c_k(\vec{V}_{c_k,i})$$

are the labels of vertices in P. All new variables introduced here are considered to be fresh.

From the definition of valid ranking, the definition of $r(\cdot)$ above, and the definition of connector set it follows that

$$\{A, B\} \setminus I \neq \emptyset \Rightarrow \min\{r(t_1) \mid t_1 \in \{A, B\} \setminus I\} > \max\{r(t_2) \mid t_2 \in N(A, B, P)\}$$

for any path P from a to b.

Each **Resolution** step either completes the given branch of the algorithm or creates several new branches. In order to show the algorithm termination, it is sufficient to demonstrate that all branches terminate. Consider any single branch. Consider the sets

$$S_i = (\text{all type expressions in } T'_i) \setminus I$$

for T'_i that occur on i-th step in the branch under consideration. Each step of the algorithm either adds one or more of the following to S_i to produce S_{i+1} or terminates its branch:

1. If $\text{lub}_i\, w_i = w \in S_i$, add $\{w_i\} \setminus I$ to S_{i+1};
2. If $\text{glb}_i\, w_i = w \in S_i$, add $\{w_i\} \setminus I$ to S_{i+1};
3. If $a(\vec{A}) = A \in S_i$, $i \in I$ and P is a path from a to i in $\mathcal{G}$, add $N(A, i, P)$ to S_{i+1};
4. If $a(\vec{A}) = A \in S_i$, $i \in I$ and P is a path from i to a in $\mathcal{G}$, add $N(i, A, P)$ to S_{i+1};
5. If $a(\vec{A}) = A \in S_i$, $b(\vec{B}) = B \in S_i$, and P is a path from a to b in $\mathcal{G}$, add $N(A, B, P)$ to S_{i+1}.

Thus, each transition from S_i to S_{i+1} is a reproduction step of the evolution system $\mathcal{E} = \langle E, R, I, r(\cdot), S_0 \rangle$, where E is the set of all possible type expressions. Reproduction steps never repeat in a single branch due to the presence of the set of already processed constraints D. Therefore, the conditions of Lemma 5.1 are satisfied and the branch under consideration terminates. Due to the arbitrary choice of the branch, this proves the algorithm termination.

□

5.1.4 Termination of entailment algorithm

Algorithm 4.2 is used to decide entailment on type constraint sets. It transforms the task of deciding entailment into the question of possibility of flattening. The transformation is done by creating a new user type graph that is obtained from the original one by adding certain vertices and edges to it. Since flattening algorithm terminates whenever the user type graph has a valid ranking (Theorem 5.2) and Algorithm 4.2 always checks whether a newly generated graph has a valid ranking before it makes an attempt to flatten against it, the following theorem has been proven:

Theorem 5.3 (Termination of entailment algorithm). If the graph $\mathcal{G}$ has a valid ranking then Algorithm 4.2 terminates. □

Thus, the termination of all non-trivial algorithms described in Chapter 4 has been proven. The next section formally introduces the notions related to types and type constraints and applies them to establish correctness properties of the main algorithms. These properties will be used in the proof of subject reduction theorem in Section 5.3.

5.2 Types

In this section, the basic notions of the type system theory will be formally defined. These notions are:

1. Ground types represented as regular trees over a finite alphabet (a set of type constructors). These will be introduced in Section 5.2.1.
2. Subtyping defined as a partial pre-order on the domain of ground types. Subtyping is defined in Section 5.2.2.
3. Entailment as a relationship between sets of subtyping constraints will be introduced in Section 5.2.3.
4. Constrained types that denote certain sets over the domain of ground types will be defined in Section 5.2.5. The notion of subtyping as well as other notions presented so far will be extended to deal with constrained types.

In Section 5.2.4 these notions will be used to prove that the entailment and flattening algorithms of Chapter 4 are correct with respect to the regular tree model developed so far. The flattening algorithm is also proven to be complete.

5.2.1 Regular trees

The model that is used for type theory developed in this dissertation is the model of *regular trees* defined below.

Definition 5.3 (Regular trees). Let L be a finite ranked alphabet $L = \{a_i\}_i$ with a (total) ranking function $\mathit{Arity}(\cdot) : L \rightarrow (\mathbb{N} \cup \{0\})$. Then a *tree* ω *over* L is a partial function $\omega : \mathbb{N}^* \rightarrow L$ from sequences δ of natural numbers into elements of L that satisfies the following conditions:

1. $\langle\rangle \in \mathrm{dom}\,\omega$, where $\langle\rangle$ denotes the empty sequence;
2. If $\omega(\delta) = a_i$ then $\omega(\delta j)$ is defined for all $j \in \{1, \ldots, \mathit{Arity}(a_i)\}$ and undefined for all $j > \mathit{Arity}(a_i)$;
3. If $\omega(\delta j)$ is defined, then $\omega(\delta)$ is also defined.

A *subtree* τ of a tree ω at δ where $\omega(\delta)$ exists is defined as

$$\tau(\delta') \stackrel{\text{def}}{=} \omega(\delta\delta')$$

A *regular tree* is a tree that has a finite number of subtrees. The domain of all regular trees over L is denoted $\mathit{Tree}(L)$. A regular tree ω that has an additional property that

$$\exists N\colon\ \ \forall \delta \in \mathbb{N}^*, |\delta| > N\colon\ \ \omega(\delta) \text{ is undefined}$$

is called *a finite tree*. The domain of all finite trees over L is denoted as $\mathit{Tree}_{\mathit{fin}}(L)$. □

Definition 5.4 (Contractive regular system of equations). Let L be a ranked alphabet and TyVar be a set of (0-ary) *type variables* ($\mathit{TyVar} \cap L = \emptyset$). Then

$$\alpha_i = e_i(\alpha_1, \ldots, \alpha_n) \tag{5.6}$$

where $\alpha_i \in \mathit{TyVar}$, $e_i(\alpha_1, \ldots, \alpha_n) \in \mathit{Tree}_{\mathit{fin}}(L \cup \{\alpha_1, \ldots, \alpha_n\})$ is called a *regular system of equations over* $\mathit{Tree}(L)$. If, in addition to the above, $e_i(\alpha_1, \ldots, \alpha_n) \notin \mathit{TyVar}$ for all i, then (5.6) is called a *contractive regular system of equations.* A tuple $\langle t_1, \ldots, t_n\rangle$ is called *a solution* of (5.6) iff

$$t_i = e_i[t_1/\alpha_1, \ldots, t_n/\alpha_n](\cdot)$$

where

$$e[a/\alpha](\delta) \stackrel{\text{def}}{=} \begin{cases} a(\delta'') & \text{if } \delta = \delta'\delta'' \text{ and } e(\delta') = \alpha \\ e(\delta) & \text{otherwise} \end{cases}$$

□

Algebraic and topological properties of the domain $\mathit{Tree}(L)$ have been studied in [AN80] and [Cou83]. Properties of systems of contractive regular equations over such domains have been investigated in [Cou86]. Regular trees were also used to model recursive types in [AC93]. The following result from the abovementioned papers will be used later in this chapter.

Theorem 5.4 (Contractive regular system solution). If E is a contractive regular system of equations over $\mathit{Tree}(L)$, then E has a unique solution in $\mathit{Tree}(L)$. □

In [AC93] it is also shown that $\mathit{Tree}_{\mathit{fin}}(L)$ is isomorphic to the domain of *finite ground types* (*fgt*) defined by the grammar

$$\mathit{fgt} ::= a(\mathit{fgt}_1, \ldots, \mathit{fgt}_{\mathit{Arity}(a)})$$

where $a \in L$, while the domain of *recursive ground types* (*rgt*) defined by the grammar

$$\mathit{rgt} ::= \alpha | a(\mathit{rgt}_1, \ldots, \mathit{rgt}_{\mathit{Arity}(a)}) | \mu\alpha.\mathit{rgt}$$

(where $\alpha \in \mathit{TyVar}$, $a \in L$) is a subdomain of $\mathit{Tree}(L)$. From now on, the above relationships between tree and type domains will be used to give meaning to the usage of types in contexts requiring trees and vise versa.

5.2.2 Subtyping

This section defines subtyping as a partial pre-order over the domain of ground pre-types which are regular trees satisfying certain conditions. Subtyping is defined with respect to the *type graph* that is given by the set of constructors, their arities and variances, and subtyping relationships between them. These subtyping relationships are expressed as *constraint systems* that are defined below. This way of defining subtyping relations is due to [Pot98], but is generalized here to allow significantly more flexibility in the definition of the basic type system.

Definition 5.5 (Constraint system). Let L be a ranked alphabet and $TyVar$ be a set of (0-ary) *tree variables* ($TyVar \cap L = \emptyset$). Then

$$\bigwedge_i (\underline{e}_i(\alpha_1, \dots, \alpha_n) \preceq \overline{e}_i(\alpha_1, \dots, \alpha_n))$$

where $\alpha_i \in TyVar$, $e_i(\alpha_1, \dots, \alpha_n) \in Tree_{fin}(L \cup \{\alpha_1, \dots, \alpha_n\})$ is called a *constraint system over* $Tree(L)$ (denoted $\mathbb{C}$). An *empty constraint system* with free variables $\alpha_1, \dots, \alpha_n$ is denoted $TRUE(\alpha_1, \dots, \alpha_n)$.

□

For example, both $TRUE(\alpha)$ and $\alpha \preceq c(a \to \alpha) \wedge \alpha \preceq \beta$ are constraint systems.

Definition 5.6 (Constraint formula). A logical formula $\mathbb{F}$ built with logical connectives $\wedge$ and $\vee$ is called *a constraint formula over* L iff every atom in $\mathbb{F}$ has one of the forms:

$$\underline{e}_i(\alpha_1, \dots, \alpha_n) \preceq \overline{e}_i(\alpha_1, \dots, \alpha_n)$$

or

$$TRUE(\alpha_1, \dots, \alpha_n)$$

or

$$FALSE(\alpha_1, \dots, \alpha_n)$$

where $\alpha_i \in TyVar$ and $e_i(\alpha_1, \dots, \alpha_n) \in Tree_{fin}(L \cup \{\alpha_1, \dots, \alpha_n\})$ □

Each constraint system is a constraint formula. Moreover, each constraint formula can be represented in the form $\vee_i \mathbb{C}_i$ where each of $\mathbb{C}_i$ is a constraint system.

The satisfiability of a constraint formula (system) can only be defined with respect to the semantics of a given subtyping relationship $\preceq$ (defined below).

Definition 5.7 (Type graph). A *type graph* $\mathcal{G}$ is a directed acyclic graph with vertices from a ranked alphabet $L = L(\mathcal{G})$ (with ranking function $Arity(\cdot)$) and marked edges together with *variance function* $Var(\cdot, \cdot) : L, \mathbb{N} \to \{+, -, 0\}$ such that $Var(a, i)$ is defined for all i from 1 to $Arity(a)$. Edges of the graph $\underline{a} \to \overline{a}$ are marked with constraint systems $\mathbb{C}$ over L such that

$$FTV(\mathbb{C}) = \{\underline{\alpha}_1, \dots, \underline{\alpha}_{Arity(\underline{a})}, \overline{\alpha}_1, \dots, \overline{\alpha}_{Arity(\overline{a})}\}$$

□

The following defines ground pre-types as regular trees with no infinite sequences of bound operators (lub and glb).

Definition 5.8 (Ground pre-types). If $\mathcal{G}$ is a type graph, then the domain of *ground pre-types* $PreTyp(\mathcal{G})$ *over* $\mathcal{G}$ is defined as the domain of regular trees over $L(\mathcal{G}) \cup \{\text{glb}, \text{lub}\}$ where $Arity(\text{glb}) = Arity(\text{lub}) = 2$ restricted by the following condition: if ω is a pre-type and $\{j_i\}_{i=1}^{\infty}$ is an infinite sequence such that $\forall i$: $\langle j_1, \dots, j_i \rangle \in \text{dom}(\omega)$, then $\forall n \exists i > n$: $\omega(\langle j_1, \dots, j_i \rangle) \in L$. □

The last condition is designed to eliminate the "nonsense" trees like lub(lub(lub(...)), lub(lub(...))).

Subtyping is defined below as a binary relation on ground pre-types. It is defined as the limit of the infinite series of decreasing relations $\preceq_1, \ldots, \preceq_n, \ldots$. These relations are defined recursively below.

Definition 5.9 (*k*-subtyping). *k-subtyping* (for $k \in \mathbb{N} \cup \{0\}$) is a binary relationship over $PreTyp(\mathcal{G}) \times PreTyp(\mathcal{G})$ (denoted $\preceq_k$) defined recursively as follows:

0-subtyping: $\underline{\omega} \preceq_0 \overline{\omega}$ for all $\underline{\omega}, \overline{\omega}$

Structural: $a(\underline{\omega}_1, \ldots, \underline{\omega}_{Arity(a)}) \preceq_k a(\overline{\omega}_1, \ldots, \overline{\omega}_{Arity(a)}) \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{i=1}^{Arity(a)} f_i^a(\underline{\omega}_i, \overline{\omega}_i)$

$$\text{where } a \in L(\mathcal{G}) \text{ and } f_i^a(\underline{\omega}, \overline{\omega}) = \begin{cases} \underline{\omega} \preceq_{k-1} \overline{\omega} & \text{if } Var(a,i) = + \\ \overline{\omega} \preceq_{k-1} \underline{\omega} & \text{if } Var(a,i) = - \\ \underline{\omega} \preceq_{k-1} \overline{\omega} \wedge \overline{\omega} \preceq_{k-1} \underline{\omega} & \text{if } Var(a,i) = 0 \end{cases}$$

Constructors:

$$\underline{a}(\underline{\omega}_1, \ldots, \underline{\omega}_{Arity(\underline{a})}) \preceq_k \overline{a}(\overline{\omega}_1, \ldots, \overline{\omega}_{Arity(\overline{a})})$$

$$\overset{\text{def}}{\Longleftrightarrow} \bigvee_p \exists \omega_1, \ldots, \omega_{n_p}: \ \bigwedge_{i=1}^{|p|} C_i^p[\preceq_{k-1}][\omega_1, \ldots, \omega_{n_p}]$$

where $\underline{a} \neq \overline{a}$, $p = \langle \underline{a} = a_0, a_1, \ldots . a_{|p|} = \overline{a} \rangle$ is a path from $\underline{a}$ to $\overline{a}$ in $\mathcal{G}$ with edges labeled with $\mathbb{C}_i^p$ $(i \in [1, \ldots, |p|])$, $n_p = |\cup_{i=0}^{|p|} FTV(C_i^p)|$, and C_i^p is defined as

Case $i = 0$:

$$C_0^p \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{j=1}^{Arity(a_0)} f_j^{a_0}(\underline{\omega}_j, \overline{\alpha}_j^0)$$

Case $i \in [1, \ldots, |p|]$:

$$C_i^p \overset{\text{def}}{\Longleftrightarrow} \bigwedge_{j=1}^{Arity(a_0)} f_j^{a_i}(\underline{\alpha}_j^i, \overline{\alpha}_j^i)$$
$$\wedge\, \mathbb{C}_i^p[\overline{\alpha}_1^{i-1}/\underline{\alpha}_1, \ldots . \overline{\alpha}_{Arity(a_{i-1})}^{i-1}/\underline{\alpha}_{Arity(a_{i-1})}, \underline{\alpha}_j^i/\overline{\alpha}_1, \ldots, \underline{\alpha}_{Arity(a_i)}^i/\overline{\alpha}_{Arity(a_i)}]$$

where $\overline{\alpha}_i^{|p|} \equiv \overline{\omega}_i$, and $\underline{\alpha}_j^i$ and $\overline{\alpha}_j^i$ $(i \neq |p|)$ are fresh type variables.

Lub:

$$\text{lub}(\underline{\omega}_1, \underline{\omega}_2) \preceq_k \overline{\omega} \overset{\text{def}}{\Longleftrightarrow} \underline{\omega}_1 \preceq_k \overline{\omega} \wedge \underline{\omega}_2 \preceq_k \overline{\omega}$$

and

$$\underline{\omega} \preceq_k \text{lub}(\overline{\omega}_1, \overline{\omega}_2) \overset{\text{def}}{\Longleftrightarrow} \underline{\omega} \preceq_k \overline{\omega}_1 \vee \underline{\omega} \preceq_k \overline{\omega}_2$$

Glb:

$$\text{glb}(\underline{\omega}_1, \underline{\omega}_2) \preceq_k \overline{\omega} \overset{\text{def}}{\Longleftrightarrow} \underline{\omega}_1 \preceq_k \overline{\omega} \vee \underline{\omega}_2 \preceq_k \overline{\omega}$$

and

$$\underline{\omega} \preceq_k \text{glb}(\overline{\omega}_1, \overline{\omega}_2) \overset{\text{def}}{\Longleftrightarrow} \underline{\omega} \preceq_k \overline{\omega}_1 \wedge \underline{\omega} \preceq_k \overline{\omega}_2$$

□

The series $\preceq_k$ is a decreasing sequence of symmetric and transitive relations on $PreTyp(\mathcal{G})$. This fact immediately follows from case analysis of Definition 5.9.

Definition 5.10 (Subtyping). *Subtyping* is a binary relation over $PreTyp(\mathcal{G}) \times PreTyp(\mathcal{G})$ (denoted $\preceq$) defined as

$$\underline{\omega} \preceq \overline{\omega} \overset{\text{def}}{\Longleftrightarrow} \forall k: \ \underline{\omega} \preceq_k \overline{\omega}$$

□

Subtyping is a symmetric and transitive relationship on $PreTyp(\mathcal{G})$.

Definition 5.10 defines a generalization of the traditional notion of subtyping of recursive type trees for the case of variance-annotated, non-structural graph $\mathcal{G}$ with conditional subtyping.

Definition 5.10 does not state that subtyping should be a partial order, it is only required to be a pre-order. Indeed, $\mathrm{lub}(a,a) \preceq a$ and $a \preceq \mathrm{lub}(a,a)$ while $a \not\equiv \mathrm{lub}(a,a)$ as trees. Given a type graph $\mathcal{G}$, the domain of types $Typ(\mathcal{G})$ is defined as $(PreTyp(\mathcal{G})/=)$ where $(\omega_1 = \omega_2) \overset{\text{def}}{\Longleftrightarrow} (\omega_1 \preceq \omega_2 \wedge \omega_2 \preceq \omega_1)$. From now on, equality $(\omega_1 = \omega_2)$ will be used to denote the relationship just defined, while equivalence $(\omega_1 \equiv \omega_2)$ will be used to denote equality between ω_1 and ω_2 as regular trees. The equivalence classes of ground pre-types will be called *ground types* and will be denoted (with a slight abuse of notation) by their representatives. For example, both a and $\mathrm{lub}(a,a)$ denote the same ground type. The subtyping relationship over $PreTyp(\mathcal{G})$ induces partial order (subtyping) over $Typ(\mathcal{G})$.

5.2.3 Entailment

The entailment relationship between constraint systems introduced in this section is central for the presented type theory. Entailment expresses the fact that one constraint system (formula) follows from the other in the sense of first-order logic.

Definition 5.11 (Entailment). If $\overline{\mathbb{C}}[\overline{\alpha}_1, \ldots, \overline{\alpha}_{\overline{N}}]$ and $\underline{\mathbb{C}}[\underline{\alpha}_1, \ldots, \underline{\alpha}_{\underline{N}}, \overline{\alpha}_1, \ldots, \overline{\alpha}_{\overline{N}}]$ are constraint formulae over $L(\mathcal{G}) \cup \{\mathrm{lub}, \mathrm{glb}\}$, then $\overline{\mathbb{C}}$ *implies* $\underline{\mathbb{C}}$ *w.r.t.* $\mathcal{G}$ (denoted $\overline{\mathbb{C}} \vdash_{\mathcal{G}} \underline{\mathbb{C}}$) iff

$$\forall \overline{\tau}_1, \ldots, \overline{\tau}_{\overline{N}} \in PreTyp(\mathcal{G}): \ \overline{\mathbb{C}}[\overline{\tau}_1/\overline{\alpha}_1, \ldots, \overline{\tau}_{\overline{N}}/\overline{\alpha}_{\overline{N}}] \Rightarrow \exists \underline{\tau}_1, \ldots, \underline{\tau}_{\underline{N}} \in PreTyp(\mathcal{G}):$$
$$\underline{\mathbb{C}}[\underline{\tau}_1/\underline{\alpha}_1, \ldots, \underline{\tau}_{\underline{N}}\underline{\alpha}_{\underline{N}}, \overline{\tau}_1/\overline{\alpha}_1, \ldots, \overline{\tau}_{\overline{N}}/\overline{\alpha}_{\overline{N}}]$$

□

For example,

$$\alpha \preceq \beta \wedge \beta \preceq \gamma \vdash \alpha \preceq \gamma$$

is a true statement since for any α and γ such that $\alpha \preceq \beta$ and $\beta \preceq \gamma$, $\alpha \preceq \gamma$ also holds. Another example of a true statement is

$$\alpha \preceq \textbf{T\_Integer} \vdash_{\mathcal{G}} \beta \preceq \textbf{T\_Real} \wedge \alpha \preceq \beta$$

(provided that **T_Integer** $\leq_{\mathcal{G}}$ **T_Real**). Indeed, for any α that is less than **T_Integer** it is possible to take $\beta = \alpha$ which will automatically satisfy

$$\beta \preceq \textbf{T\_Real} \wedge \alpha \preceq \beta$$

An example of a false entailment is

$$TRUE(\alpha) \vdash \alpha \preceq \textbf{T\_Integer}$$

since not every type is a subtype of **T_Integer**. Note however that $\vdash \alpha \preceq$ **T_Integer** is a true statement as there *exists* an α such that $\alpha \preceq$ **T_Integer**.

The following are the properties of the entailment relationship defined above. All of them trivially follow from Definition 5.11.

Reflexivity: $\mathbb{C} \vdash_{\mathcal{G}} \mathbb{C}$ for any constraint set $\mathbb{C}$

False premises: If $\not\vdash_{\mathcal{G}} \mathbb{C}_a$ then $\mathbb{C}_a \vdash_{\mathcal{G}} \mathbb{C}_b$ for any $\mathbb{C}_b$

Right elimination: If $\mathbb{C}_a \vdash_{\mathcal{G}} \mathbb{C}_b \wedge \mathbb{C}_c$ then $\mathbb{C}_a \vdash_{\mathcal{G}} \mathbb{C}_b$

Restricted transitivity: If $\mathbb{C}_a \vdash_{\mathcal{G}} \mathbb{C}_b \wedge \mathbb{C}_c$ and $\mathbb{C}_c \vdash_{\mathcal{G}} \mathbb{C}_d$, then $\mathbb{C}_a \vdash_{\mathcal{G}} \mathbb{C}_b \wedge \mathbb{C}_c \wedge \mathbb{C}_d$ provided that

$$(FTV(\mathbb{C}_d) \setminus FTV(\mathbb{C}_c)) \cap (FTV(\mathbb{C}_a) \cup FTV(\mathbb{C}_b)) = \emptyset$$

The following lemma establishes the relationship between the constraints on the left-hand side of the turnstyle and the set of constraints implicit in the construction of the graph $\mathcal{G}$.

Lemma 5.2 (Extended graph entailment). If $\mathcal{G}$ is a type graph, $\mathbb{C}^1$ has a solution in *PreTyp*($\mathcal{G}$), and $\mathbb{C}^2[a_i/\alpha_i]$ has a solution in *PreTyp*($\mathcal{G}_e$), then $\mathbb{C}^1 \vdash_{\mathcal{G}} \mathbb{C}^2$, where $\mathcal{G}_e$ is defined as follows:

1. $\mathcal{G} \subseteq \mathcal{G}_e$
2. For every type variable $\alpha_i \in FTV(\mathbb{C}^1)$ there is a 0-ary type constructor $a_i \in \mathcal{G}_e$ such that $a_i \notin L(\mathcal{G})$
3. For every constraint $c = (T_1 \preceq T_2)$ in $\mathbb{C}^1$, there is an edge from *pfunctor*($T_1[a_i/\alpha_i]$) to *pfunctor*($T_2[a_i/\alpha_i]$) in $\mathcal{G}_e$ labeled with $c[a_i/\alpha_i]$.

□

Proof. Assume M^1 is a solution of $\mathbb{C}^1$ in *PreTyp*($\mathcal{G}$) and M^2 is a solution of $\mathbb{C}^2$ in *PreTyp*($\mathcal{G}_e$). It will be shown that $M = M^2[M^1(\alpha_i)/a_i] \cup M^1$ is a solution of $\mathbb{C}^2[\alpha_i/a_i]$ in *PreTyp*($\mathcal{G}$). Once this is done, the entailment $\mathbb{C}^1 \vdash_{\mathcal{G}} \mathbb{C}^2$ will follow directly (as it will follow that for any solution M^1 of $\mathbb{C}^1$ there exists a solution M of $\mathbb{C}^2$ such that $M^1 \subseteq M$). In order to show that M is a solution of $\mathbb{C}^2[\alpha_i/a_i]$ in *PreTyp*($\mathcal{G}_e$), it is sufficient to prove that

$$\underline{t} \preceq_{\mathcal{G}_e} \bar{t} \Rightarrow \underline{t}[M^1(\alpha_i)/a_i \text{ for all } i] \preceq_{\mathcal{G}} \bar{t}[M^1(\alpha_i)/a_i \text{ for all } i] \tag{5.7}$$

for all $\underline{t}, \bar{t} \in$ *PreTyp*($\mathcal{G}_e$). Indeed, if (5.7) is satisfied, then

$$\mathbb{C}^2[\alpha_i/a_i][M] \equiv \mathbb{C}^2[\alpha_i/a_i][M^2[M^1(\alpha_i)/a_i] \cup M^1]] \equiv \mathbb{C}^2[M^2][M^1(\alpha_i)/a_i]$$

which follows from the fact that M^2 is a solution of $\mathbb{C}^2$ in *PreTyp*($\mathcal{G}_e$) and from (5.7).

The proof of (5.7) is by induction.

Basis:

$$\underline{t} \preceq_{\mathcal{G}_e} \bar{t} \Rightarrow \underline{t}[M^1(\alpha_i)/a_i \text{ for all } i] \preceq_{0,\mathcal{G}} \bar{t}[M^1(\alpha_i)/a_i \text{ for all } i]$$

is trivially satisfied since $x \preceq_{0,\mathcal{G}} y$ is true for all x and y.

Induction step: From k to $k+1$: If

$$\underline{t} \preceq_{\mathcal{G}_e} \bar{t} \Rightarrow \underline{t}[M^1(\alpha_i)/a_i \text{ for all } i] \preceq_{k,\mathcal{G}} \bar{t}[M^1(\alpha_i)/a_i \text{ for all } i] \tag{5.8}$$

then

$$\underline{t} \preceq_{\mathcal{G}_e} \bar{t} \Rightarrow \underline{t}[M^1(\alpha_i)/a_i \text{ for all } i] \preceq_{k+1,\mathcal{G}} \bar{t}[M^1(\alpha_i)/a_i \text{ for all } i] \tag{5.9}$$

There are several possible cases:

1. *pfunctor*($\underline{t}$) = *pfunctor*($\bar{t}$) $\in L(\mathcal{G}_e) \setminus L(\mathcal{G})$. In this case, $\underline{t} = \bar{t} = a_i$ and the subtyping is immediate for all k including $k+1$.

2. *pfunctor*($\underline{t}$) = *pfunctor*($\overline{t}$) $\in L(\mathcal{G})$. According to the **Structural** clause of the subtyping definition,

$$\underline{t}[M^1(\alpha_i)/a_i \text{ for all } i] \preceq_{k+1,\mathcal{G}} \overline{t}[M^1(\alpha_i)/a_i \text{ for all } i]$$
$$\equiv \bigwedge_{i=1}^{n_a} f_i^{a,\mathcal{G}}[\preceq_k](\underline{\omega}_i, \overline{\omega}_i)[M^1(\alpha_i)/a_i \text{ for all } i]$$

where $n_a = Arity(a)$, $\underline{t} = a(\underline{\omega}_1, \ldots, \underline{\omega}_{n_a})$ and $\overline{t} = a(\overline{\omega}_1, \ldots, \overline{\omega}_{n_a})$. On the other hand, since $\underline{t} \preceq_{\mathcal{G}_e} \overline{t}$, the following is true:

$$\bigwedge_{i=1}^{n_a} f_i^{a,\mathcal{G}_e}[\preceq_m](\underline{\omega}_i, \overline{\omega}_i) \qquad \text{for all } m$$

and therefore (by induction assumption for k) $\bigwedge_{i=1}^{n_a} f_i^{a,\mathcal{G}}[\preceq_k](\underline{\omega}_i, \overline{\omega}_i)[M^1(\alpha_i)/a_i$ for all $i]$.

3. *pfunctor*($\underline{t}$) $\neq$ *pfunctor*($\overline{t}$), $\{$*pfunctor*($\underline{t}$), *pfunctor*($\overline{t}$)$\} \subseteq L(\mathcal{G}_e)$. According to the clause **Constructors** of the subtyping definition and to the assumption $\underline{t} \preceq_{\mathcal{G}_e} \overline{t}$, there exists a path p from $\underline{a}$ = *pfunctor*($\underline{t}$) to $\overline{a}$ = *pfunctor*($\overline{t}$) in $\mathcal{G}_e$ such that $|p| > 1$ and

$$\exists \omega_1, \ldots, \omega_{n_p}: \quad \bigwedge_{j=1}^{|p|} C_j^p[\preceq_{\mathcal{G}_e}][\omega_1, \ldots, \omega_{n_p}]$$

where C_j^p are defined as in the definition of subtyping. This is equivalent to

$$\exists\{\alpha_m^j\}: \quad \bigwedge_{j=2}^{|p|-1} a_{j-1}^p(\alpha_1^{j-1}, \ldots, \alpha_{Arity(a_{j-1}^p)}^{j-1}) \preceq_{\mathcal{G}_e} a_j^p(\alpha_1^j, \ldots, \alpha_{Arity(a_j^p)}^j)$$
$$\wedge\, \underline{t} \preceq_{\mathcal{G}_e} a_1^p(\alpha_1^1, \ldots, \alpha_{Arity(a_1^p)}^1) \wedge a_{|p|-1}^p(\alpha_1^{|p|-1}, \ldots, \alpha_{Arity(a_{|p|-1}^p)}^{|p|-1}) \preceq_{\mathcal{G}_e} \overline{t} \quad (5.10)$$

where $\alpha_m^j \in PreTyp(\mathcal{G}_e)$ and for each j, a_j^p is a type constructor located in j-th position of the path p. By induction assumption

$$\exists \omega_1, \ldots, \omega_{n_p}: \quad \bigwedge_{j=1}^{|p|} C_j^p[\preceq_{k,\mathcal{G}}][\omega_1, \ldots, \omega_{n_p}][M^1(\alpha_i)/a_i \text{ for all } a_i] \quad (5.11)$$

In order to show (5.9), it is sufficient to demonstrate that

$$\exists\{\alpha_m^j\}: \quad (\bigwedge_{j=2}^{|p'|-1} a_{j-1}^{p'}(\alpha_1^{j-1}, \ldots, \alpha_{Arity(a_{j-1}^{p'})}^{j-1}) \preceq_{k+1,\mathcal{G}} a_j^{p'}(\alpha_1^j, \ldots, \alpha_{Arity(a_j^{p'})}^j)$$
$$\wedge\, \underline{t} \preceq_{k+1,\mathcal{G}} a_1^{p'}(\alpha_1^1, \ldots, \alpha_{Arity(a_1^{p'})}^1)$$
$$\wedge\, a_{|p'|-1}^{p'}(\alpha_1^{|p'|-1}, \ldots, \alpha_{Arity(a_{|p'|-1}^{p'})}^{|p'|-1}) \preceq_{k+1,\mathcal{G}} \overline{t})[M^1(\alpha_i)/a_i \text{ for all } i] \quad (5.12)$$

where $p' = p \cap L(\mathcal{G})$. This will be done by separately considering each constraint. Each constraint in (5.10) corresponds to a single edge $e_j = a_{j-1}^p \rightarrow a_j^p$ in the path p. Each constraint in (5.12) corresponds to a single edge $e_j = a_{j-1}^{p'} \rightarrow a_j^{p'}$ in the path p' which corresponds to a segment of the path p. There are several possible cases:

(a) $e_j \in \mathcal{G}$. In this case, the constraint is immediately satisfied by (5.11)

(b) $e_j \notin \mathcal{G}$, $\{a_{j-1}, a_j\} \subseteq L(\mathcal{G})$. By construction of $\mathcal{G}_e$, constraint C marking this edge in $\mathcal{G}_e$ has the following properties: $FTV(C) = \emptyset$ and $C[M^1(\alpha_i)/a_i$ for all $i]$ is true. From this, assumption (5.11), and the construction of C_j^p

$$C_j^p[\preceq_{k,\mathcal{G}}][M^1(\alpha_i)/a_i \text{ for all } i]$$

From the above and the definition of subtyping

$$(a_{j-1}^p(\alpha_1^{j-1}, \ldots, \alpha_{Arity(a_{j-1}^p)}^{j-1}) \preceq_{k+1,\mathcal{G}} a_j^p(\alpha_1^j, \ldots, \alpha_{Arity(a_j^p)}^j))[M^1(\alpha_i)/a_i \text{ for all } i]$$

(c) $a_{j-1} \notin L(\mathcal{G}) \vee a_j \notin L(\mathcal{G})$. In this case, consider the minimal segment s of the path p such that:

$$e_j \in s$$
$$e_{j_m} \in s \wedge a_{j_m} \notin L(\mathcal{G}) \Rightarrow e_{j_m+1} \in s$$
$$e_{j_m} \in s \wedge a_{j_m-1} \notin L(\mathcal{G}) \Rightarrow e_{j_m-1} \in s$$

Let s have the length M'. Then all $M'-1$ vertices inside s are the "additional" type constructors a_i and therefore do not belong to p'. All constraints marking the internal edges of s therefore have the form $\underline{a}_i \preceq \overline{a}_i$ and are satisfied in

$$(\underline{a}_i \preceq_{\mathcal{G}} \overline{a}_i)[M^1(\alpha_i)/a_i \text{ for all } i]$$

by construction of the graph $\mathcal{G}_e$ and the solution M^1. Since all additional constructors are 0-ary, this also implies that $C_j^p[\preceq_{k,\mathcal{G}}][M^1(\alpha_i)/a_i$ for all $i]$ for all internal edges. For the external edges, $C_j^p[\preceq_{k,\mathcal{G}}][M^1(\alpha_i)/a_i$ for all $i]$ is satisfied by (5.11). Thus,

$$(a_{j-1}^{p'}(\alpha_1^{j-1}, \ldots, \alpha_{Arity(a_{j-1}^{p'})}^{j-1}) \preceq_{k+1,\mathcal{G}} a_j^{p'}(\alpha_1^j, \ldots, \alpha_{Arity(a_j^{p'})}^j))[M^1(\alpha_i)/a_i \text{ for all } i]$$

where $a_{j-1}^{p'}$ is the first and $a_j^{p'}$ is the last constructor in s. Therefore (5.11) is satisfied.

4. Otherwise, $\{pfunctor(\underline{t}), pfunctor(\overline{t})\} \cap \{\text{lub}, \text{glb}\} \neq \emptyset$. Since the clauses **Lub** and **Glb** are independent of the user type graph and the trees under consideration are in $PreTyp(\mathcal{G}_e)$, the subtyping clause under consideration can be equivalently formulated as

$$\bigwedge_{j=1}^{N} \underline{A_j} \preceq_{k+1,\mathcal{G}} \overline{A_j}$$

where

$$\{pfunctor(\underline{A_j}), pfunctor(\overline{A_j})\} \cap \{\text{lub}, \text{glb}\} = \emptyset$$

For each of these subtyping constraints (5.9) holds by one of the previously considered cases, and therefore (5.9) holds for this case as well.

□

In this section, the notion of entailment was introduced and its properties were studied. In the following section, correctness and completeness of the flattening algorithm and the correctness of the entailment algorithm will be demonstrated.

5.2.4 Properties of entailment and flattening algorithms

Before the algorithm properties can be established, it is necessary to specify how the user type graph defined by the program according to Definition 4.6 can be translated into the type graph defined and used in this section (Definition 5.7).

Let $\mathcal{G}_u$ be a user type graph (defined by Definition 4.6). The type graph $\mathcal{G}$ is constructed from $\mathcal{G}_u$ as follows:

1. For each type definition in $\mathcal{G}_u$, there is a type constructor in $L(\mathcal{G})$ (and, respectively, a vertex in $\mathcal{G}$) with arity and variance defined by the type definition
2. For each edge in $\mathcal{G}_u$ from $\underline{a}$ to $\overline{a}$ marked with constraints $\mathbb{C}$ there is a corresponding edge in $\mathcal{G}$ marked with $\mathbb{C} \wedge \mathbb{C}_{\underline{a}} \wedge \mathbb{C}_{\overline{a}}$, where $\mathbb{C}_{\underline{a}}$ and $\mathbb{C}_{\overline{a}}$ are constraints in type definitions of $\underline{a}$ and $\overline{a}$, respectively

$\mathcal{G}$ is then a type graph in terms of Definition 5.7.

Theorem 5.5 (Correctness and completeness of Algorithm 4.3(flattening)). If $\mathcal{G}_u$ is the user type graph, $\mathcal{G}$ is its corresponding type graph and $\mathbb{C}$ is a constraint system over $\mathcal{G}$, then:

1. If $\mathbb{F} = \mathit{flatten}_{\mathcal{G}}(\mathbb{C})$, then
   (a) The set of solutions of $\mathbb{C}$ is the same as the set of solutions of $\mathbb{F}$ (in $\mathit{PreTyp}(\mathcal{G})$)
   (b) Each disjunct in $\mathbb{F}$ has a solution in $\mathit{PreTyp}(\mathcal{G})$
2. If $\mathit{FAIL} = \mathit{flatten}_{\mathcal{G}}(\mathbb{C})$, then there is no solution to $\mathbb{C}$ in $\mathit{PreTyp}(\mathcal{G})$

□

Proof.

Point 1a: Follows from the definition of subtyping and the algorithm by a simple induction on all cases, since each step of the algorithm replaces a constraint by an equivalent constraint formula.

Point 1b: The formula $\mathbb{F}$ has the from

$$\mathbb{F} = \bigvee_i (\bigwedge_j (\bigwedge_{l \in L^i_j} l \preceq \alpha_j \;\wedge \bigwedge_{u \in U^i_j} \alpha_j \preceq u) \;\wedge \bigwedge_{\alpha_k \to \alpha_m \in G^i} \alpha_k \preceq \alpha_m \;\wedge \bigwedge_{\alpha_k = \alpha_m \in E^i} \alpha_k = \alpha_m) \quad (5.13)$$

where

$$\begin{gathered} \forall i, j, s \in L^i_j \cup U^i_j: \quad s \equiv a(\dots), a \in L(\mathcal{G}) \\ \forall i, j, s \in L^i_j \cup U^i_j: \quad FTV(s) \subseteq \mathit{vertices}(G^i_j) \\ \forall i: \quad G^i \text{ is acyclic} \\ \forall i \nexists n > 1, \alpha_{m_1}, \dots, \alpha_{m_n}: \quad (\forall k \in [1, \dots, n-1]: \quad (\alpha_{m_k} = \alpha_{m_{k+1}}) \in E_i) \\ \wedge \{\alpha_{m_1}, \alpha_{m_n}\} \subseteq \cup_{j, s \in L^i_j \cup U^i_j} FTV(s) \end{gathered} \quad (5.14)$$

It will be shown that each disjunct in (5.13) has a solution. Let D_i be a disjunct in (5.13). The presence of subformula

$$\bigwedge_{\alpha_k = \alpha_m \in E^i} \alpha_k = \alpha_m$$

does not change the set of solutions of D due to the condition (5.14). Thus, it is sufficient to demonstrate that

$$D = (\bigwedge_i (\bigwedge_{l \in L_i} l \preceq \alpha_i \;\wedge \bigwedge_{u \in U_i} \alpha_i \preceq u)) \;\wedge \bigwedge_{\alpha_k \to \alpha_m \in G} \alpha_k \preceq \alpha_m \quad (5.15)$$

where

$$\forall i, s \in L_i \cup U_i: \quad s \equiv a(\dots), a \in L(\mathcal{G}) \quad (5.16)$$
$$\forall i, s \in L_i \cup U_i: \quad FTV(s) \subseteq \mathit{vertices}(G) \quad (5.17)$$
$$G \text{ is acyclic} \quad (5.18)$$

admits a solution.

Let $L'_i = \bigcup_{j:\ \alpha_j \leq_G \alpha_i} L_j$ and $I' = \{i \mid L'_i \neq \emptyset\}$. Let E be the system of equations constructed as follows:

$$E = \{\alpha_i = \mathrm{lub}_{l \in L'_i}(l)[\bot/\alpha_j, j \notin I']\}_{i \in I'}$$

It will be shown that

1. E has a solution in $PreTyp(\mathcal{G})$
2. Any solution of E can be extended to form a solution of D

Let S^e be the set of all such subexpressions s^e of $\bigcup_{i \in I'} L'_i$ that $\nexists i \in I'$: $s^e = \alpha_i$. Let β_k be the set of fresh variables such that there is one variable for every subexpression $s^e \in S^e$. Then E can be equivalently rewritten as

$$E' = \begin{cases} \alpha_i = \mathrm{lub}(\beta_{k_{i,1}}, \ldots, \beta_{k_{i,n_i}}) & \text{for all } i \in I' \\ \beta_k = a_k(\beta_{k'_{k,1}}, \ldots, \beta_{k'_{k,Arity(a_k)}}) & \text{for all } k \end{cases}$$

where a_k is the primary functor of the expression s^e_k that corresponds to the variable β_k. The above system of equations is regular and contractive and therefore (by Theorem 5.4) admits a solution M'. M' is a map from the set $\{\alpha_i\}_{i \in I'} \cup \{\beta_k\}_k$ to the domain of regular trees over $L(\mathcal{G}) \cup \{\mathrm{lub}, \mathrm{glb}\}$. Let t_i be the regular trees mapped to the variables α_i by this solution. Then, they satisfy E (since E' and E are equivalent) and therefore

$$\bigwedge_{i \in I'} (t_i = \mathrm{lub}_{l \in L'_i}(l)[t_i/\alpha_i])$$

Since $l \in L'_i = \bigcup_{j:\ \alpha_j \leq_G \alpha_i} L_j$ and D is a disjunct in the resulting formula produced by Algorithm 4.3, each l has the form $a(\ldots)$, where $a \in vertices(\mathcal{G})$. Thus

$$\bigwedge_{i \in I'} t_i = \mathrm{lub}(s^i_1, \ldots, s^i_n)$$

where each s^i_j has the form $a^i_j(\ldots)$. Therefore each path in t_i consists of a (possibly infinite) number of segments such that

1. The length of each segment is less then or equal to the maximum depth of type variable in the expression $\mathrm{lub}(s^i_1, \ldots, s^i_n)$;
2. Each segment contains at least one of a^i_j.

Therefore, by Definition 5.8, $t_i \in PreTyp(\mathcal{G})$. Since this argument works for all i,

$$M' = \{\langle \alpha_i, t_i \rangle\}_{i \in I'}, \quad \forall i:\ \ t_i \in PreTyp(\mathcal{G})$$

is a solution for E.

Now it is left to prove that M' can be extended to a solution for D. Let's form the map $M = M' \cup \{\langle \alpha_i, \bot \rangle\}_{i \notin I'}$. This map is a variable assignment for all variables in D. It will be shown that M satisfies all constraints in D.

Constraints of the form $l \preceq \alpha_i$: Since $l \in L_i$, $l \in L'_i$ is true and therefore $L'_i \neq \emptyset$, thus $i \in I'$. Therefore

$$\begin{aligned} l[M(\alpha_k)/\alpha_k, \text{for all } k] \preceq \alpha_i[M(\alpha_k)/\alpha_k, \text{for all } k] = M(\alpha_i) = t_i \\ = \mathrm{lub}_{l \in L'_i}(l)[M(\alpha_k)/\alpha_k, \text{for all } k] \end{aligned}$$

which is true by definition of subtyping for lub.

Constraints of the form $\alpha_i \preceq \alpha_j$: If $i \in I'$ then $\emptyset \neq L'_i \subseteq L'_j$ and therefore $L'_j \neq \emptyset \Rightarrow j \in I'$. Thus there are three possible cases:

1. $i \notin I', j \notin I'$. Then the constraint under consideration becomes $\bot \preceq \bot$ which is trivially satisfied.
2. $i \notin I', j \in I'$. Then the constraint under consideration becomes $\bot \preceq M(\alpha_j)$ which is trivially satisfied.
3. $i \in I', j \in I'$. Then the constraint under consideration becomes

$$M(\alpha_i) = t_i = \text{lub}_{l \in L'_i}(l)[M(\alpha_k)/\alpha_k, \text{for all } k] \preceq \text{lub}_{l \in L'_j}(l)[M(\alpha_k)/\alpha_k, \text{for all } k] = t_j = M(\alpha_i)$$

which is true by the definition of subtyping and the fact that $L'_i \subseteq L'_j$.

Constraints of the form $\alpha_i \preceq u$: There are two possible cases:

1. $i \notin I'$. Then the constraint under consideration becomes $\bot \preceq u[M(\alpha_k)/\alpha_k, \text{for all } k]$ which is trivially satisfied.
2. $i \in I'$. The constraint under consideration becomes

$$M(\alpha_i) = t_i = \text{lub}_{l \in L'_i}(l)[M(\alpha_k)/\alpha_k, \text{for all } k] \preceq u[M(\alpha_k)/\alpha_k, \text{for all } k]$$

which is equivalent to

$$\bigwedge_{l \in L'_i} (l[M(\alpha_k)/\alpha_k, \text{for all } k] \preceq u[M(\alpha_k)/\alpha_k, \text{for all } k])$$

Consider a single conjunct of the above constraint

$$l[M(\alpha_k)/\alpha_k, \text{for all } k] \preceq u[M(\alpha_k)/\alpha_k, \text{for all } k])$$

where $l \in L'_i = \bigcup_{j:\ \alpha_j \leq_G \alpha_i} L_j$. Since $u \in U_i$ and the above constraint is a part of the disjunct D produced by the final step of the algorithm, there must exist a step **Process**(4) of the algorithm on which the constraint $l \preceq u$ is produced and added to N. This implies that at some point of the algorithm execution the constraint $l \preceq u$ is in the set C. Since $l = a_l(\ldots)$ and $u = a_u(\ldots)$ where $a_l, a_u \in L(\mathcal{G})$ and the algorithm has terminated, the constraint had to be processed on the step **Process**(4.f) (if $a_l = a_u$) or on step **Resolution**(4) (if $a_l \neq a_u$). The step **Process**(4.f) is parallel to the case **Structural** of the subtyping definition and thus produces the constraint set C' such that every k-solution of C' is a $(k+1)$-solution of $l \preceq u$. The step **Resolution**(4) is parallel to the case **Constructors** of the subtyping definition and therefore it also produces the set of constraints C' such that every k-solution of C' is a $k+1$-solution of $l \preceq u$. Since each step of the algorithm produces equivalent constraint sets, and all steps of the algorithm except **Resolution**(4) and **Process**(4.f) produce k-equivalent constraint sets for all k, the following statement is true:
if M_D is a k-solution of D, it is a $k+1$-solution of $l \preceq u$ and therefore (since the above argument works for all l and i) M_D is a $k+1$-solution of all constraints of the form $\alpha_i \preceq u$ in D.

Proof by induction: The solution M is a 0-solution of D (since $\preceq_0$ is universal). Assume M is a k-solution of D. Then, M is a $(k+1)$-solution of all constraints of the form $\alpha_i \preceq u$ in D. It has been shown already that M is a solution of all the constraints other then those of the form $\alpha_i \preceq u$(*). Thus, M is a $(k+1)$-solution of all constraints in D (all constraints except (*) by the statements already proven, constraints (*) by induction assumption). Therefore, M is a solution of D for all k which is equivalent to the statement that M is a solution of D.

Thus, it has been shown that D has a solution in $\mathit{PreTyp}(\mathcal{G})$.

Point 2: Follows from point 1a and the definition of subtyping, as the algorithm only fails when an unsatisfiable constraint is produced in all conjuncts of the resulting formula.

□

Thus correctness and completeness of Algorithm 4.3 have been demonstrated.

Theorem 5.6 (Correctness of Algorithm 4.2(entailment)). If $\mathcal{G}_u$ is the user type graph, $\mathcal{G}$ is its corresponding type graph and $\mathbb{C}_i^1$, $\mathbb{C}_j^2$ are constraint systems over $\mathcal{G}$, then:
If Algorithm 4.2 succeeds on input $\bigvee_i \mathbb{C}_i^1 \vdash_{\mathcal{G}} \bigvee_j \mathbb{C}_j^2$, then $\bigvee_i \mathbb{C}_i^1 \vdash_{\mathcal{G}} \bigvee_j \mathbb{C}_j^2$. □

Proof. Assume Algorithm 4.2 has succeeded. It means that for each i one and only one of the following is true:

1. Algorithm 4.3 fails while trying to flatten $\mathbb{C}_i^1$ w.r.t $\mathcal{G}$.
2. Algorithm 4.3 successfully flattens $\mathbb{C}_i^1$ w.r.t $\mathcal{G}$ and also successfully flattens at least one of $\mathbb{C}_j^2$ w.r.t $\mathcal{G}_e^i$.

In the first case, the constraint set $\mathbb{C}_i^1$ has no solutions (by Theorem 5.5) and can therefore be removed from the set of constraints on the left of the turnstyle without changing the truth value of the entailment. If this is the case for all i, the constraint formula $\bigvee_i \mathbb{C}_i^1$ has no solutions and therefore the entailment is trivially satisfied. Thus it is left to prove that

$$\bigvee_i \mathbb{C}_i^1 \vdash_{\mathcal{G}} \bigvee_j \mathbb{C}_j^2$$

under the assumption that for each i $\mathbb{C}_i^1$ has a solution in *PreTyp*($\mathcal{G}$) and there exists j_i such that $\mathbb{C}_{j_i}^2$ has a solution in *PreTyp*($\mathcal{G}_e^i$). However, the above statement follows directly from Lemma 5.2, Theorem 5.5, and the definition of entailment. □

Note that while the flattening algorithm is both correct and complete, the entailment algorithm is correct, but not complete. There are two reasons for the incompleteness. First, the extended type graph produced by the algorithm at some point might not have a valid ranking. In this case, the algorithm returns the *UNKNOWN* answer which is interpreted as *FALSE* by the type system even though the entailment might be true. Second, the formula on the left-hand side of the turnstyle is not flattened before the construction of the extended graph which leads to rejection of certain types of correct formulae. This is intentional as it makes this algorithm invariant to covariant type system transformations which play a major role in type system evolution (see Section 5.6.2).

5.2.5 Constrained types

In this section, the theory of constrained types will be developed. Constrained types are sometimes called *type schemes* (e.g. in [Pot98]).

Definition 5.12 (Simple constrained pre-type). A *simple constrained pre-type* over a graph $\mathcal{G}$ is defined as

$$\forall \alpha_1, \ldots, \alpha_n\ (\mathbb{C}).A$$

where $\mathbb{C}$ is a constraint system over $\mathcal{G}$, $FTV(\mathbb{C}) \subseteq \{\alpha_1, \ldots, \alpha_n\}$, and A is a type over $L(\mathcal{G}) \cup \{\alpha_1, \ldots, \alpha_n\}$.

□

The quantification part of the simple constrained pre-type specification will often be omitted in cases when the context determines the set of variables bound in the constrained type. For example, $\forall\alpha\ (t \preceq \alpha).\alpha$ will often be written as $(t \preceq \alpha).\alpha$.

A simple constrained pre-type can be interpreted as the lower bound of the set of ground types that it denotes. For example, if t is a ground type, then t and $(t \preceq \alpha).\alpha$ are equivalent.

Definition 5.13 (Constrained pre-type). A *constrained pre-type* X over a graph $\mathcal{G}$ is defined as follows:

$$X = \mathsf{glb}_{i=1}^{n}(\forall \alpha_1^i, \ldots, \alpha_n^i, (\mathbb{C}^i).A^i)$$

where $\forall \alpha_1^i, \ldots, \alpha_n^i, (\mathbb{C}^i).A^i$ are simple constrained pre-types. □

Thus a constrained pre-type is a greatest lower bound of a finite number of simple constrained pre-types.

Definition 5.14 (Subtyping of constrained pre-types). If

$$X_1 = \mathsf{glb}_i(\mathbb{C}_1^i).A_1^i$$
$$X_2 = \mathsf{glb}_j(\mathbb{C}_2^j).A_2^j$$

are constrained pre-types, the subtyping relationship between them is defined as follows:

$$(X_1 \preceq X_2) \overset{\text{def}}{\Longleftrightarrow} (\forall j \, \exists i: \quad \mathbb{C}_2^j \vdash_{\mathcal{G}} \mathbb{C}_1^i \wedge A_1^i \preceq A_2^j)$$

□

This subtyping relationship is transitive and reflexive on the domain of constrained pre-types.

As with types, the domain of constrained pre-types is factored by the equality relationship defined as follows:

$$(X_1 = X_2) \overset{\text{def}}{=} (X_1 \preceq X_2 \wedge X_2 \preceq X_1)$$

Thus, each constrained pre-type falls into one and only one equivalence class. These equivalence classes will be called *constrained types*. The subtyping relationship defined above is a partial order over the domain of constrained types. Again, by a slight abuse of notation, these equivalence classes will be denoted by their representatives.

A constrained pre-type with $n = 1$ is a simple constrained pre-type since $\mathsf{glb}(A) = A$. Therefore, simple constrained pre-types form a subset of constrained pre-types. Ground types can also be viewed as a subset of constrained types. Namely, if t is a ground type, a constrained type $\mathsf{glb}(().t)$ can be formed. Let t_1 and t_2 be ground types. Then

$$(\mathsf{glb}(().t_1) \preceq \mathsf{glb}(().t_2)) \equiv (\vdash_{\mathcal{G}} t_1 \preceq t_2) \equiv (t_1 \preceq t_2)$$

by the definition of subtyping and entailment.

Note that all constrained types with unsatisfiable sets of constraints are equivalent and play a role of the "top" type (denoted $\top$) in the constrained type hierarchy. Indeed, let $X = (\mathbb{C}).A$ and $X' = \mathsf{glb}_i(\mathbb{C}_i').A_i'$ where $\mathbb{C}$ is unsatisfiable. Then,

$$X' \preceq X \;\equiv\; \exists i: \quad (\mathbb{C} \vdash_{\mathcal{G}} \mathbb{C}_i' \wedge A_i' \preceq A) \;\equiv\; \mathit{TRUE}$$

for all $\mathbb{C}_i'$, A, and A_i' by the definition of entailment.

The equivalence class $\top$ therefore has "invalid" or "empty" types as its representatives. The statement that a particular (constrained) type $X = \mathsf{glb}_i(\mathbb{C}_i).A_i$ does not belong to this class will be written as

$$\vdash_{\mathcal{G}} X$$

with the interpretation

$$(\vdash_{\mathcal{G}} X \overset{\text{def}}{=} X \neq \top) \;\equiv\; (\exists i \vdash_{\mathcal{G}} \mathbb{C}_i)$$

(the last equivalence follows from the definitions of equality, subtyping, and the class $\top$).

On the other hand, a constrained type of the form $().\alpha$ plays a role of the bottom type (denoted $\bot$) since for any constrained type $X = \text{glb}_i(\mathbb{C}_i).A_i$

$$(().\alpha \preceq X) \;\equiv\; (\forall i\ \mathbb{C}_i \vdash_{\mathcal{G}} \alpha \preceq A_i) \;\equiv\; \mathit{TRUE}$$

since $\alpha \notin FTV(\mathbb{C}_i) \cup FTV(A_i)$ and thus it is possible to choose $\alpha = A_i$ to satisfy the entailment.

In order to emphasize that a subtyping relationship between two constrained types X_1 and X_2 depends on a given type graph $\mathcal{G}$, the entailment will often be written as $\vdash_{\mathcal{G}} X_1 \preceq X_2$ instead of just $X_1 \preceq X_2$.

So far all the basic notions of the type system theory have been defined and the validity of the core entailment algorithms with respect to the given interpretation of subtyping constraints has been demonstrated. In the following section, these results and notions will be used to prove the subject reduction theorem that establishes correctness of the typechecking.

5.3 Subject reduction

The subject reduction theorem is the most important result of this chapter as it proves that a successfully typechecked program does not generate type errors at run-time.

One of the preliminary results that is needed to prove subject reduction is the static subsumption theorem. It states that a more precise type specification leads to a more precise typing of the result of the computation. This result is proven in Section 5.3.1.

In order to prove the correctness of a typechecking algorithm for any language, its semantics has to be formally defined. This is done in Section 5.3.2.

The definition of "type errors" and the way reduction interacts with typing is determined by the types of the run-time objects that are processed during program execution. Questions related to run-time object typing, its definition and semantics, are considered in Section 5.3.3.

Issues related to semantics and correctness of the behavior dispatch process are formally developed in Section 5.3.4. A theorem stating that a correctly typechecked behavior application always dispatches correctly (no "message not understood" or "message ambiguous" errors) is also proven in this section.

Finally, Section 5.3.5 presents the proof of the subject reduction theorem.

5.3.1 Static subsumption

In this section, the static subsumption theorem will be proven. The intuitive meaning of this theorem is as follows:

Assume that an expression is given a type by the typing rules in Figure 4.2 under certain assumptions $\overline{\Theta}$ about the types of variables participating in the expression. Assume further that a new set of assumptions $\underline{\Theta}$ is formed in such a way that for every variable it gives the same or lesser type then the old assumption set did. Then, the theorem states, the same expression will also be typable in the new environment $\underline{\Theta}$. In addition to that, the type inferred for the expression in the new environment $\underline{\Theta}$ will be a subtype of the type inferred for the expression in the old environment $\overline{\Theta}$.

Before the proof of the static subsumption theorem can be given, two additional lemmas have to be proven. The first one simply states that different branches of type inference algorithm introduce independent sets of type variables. The next one shows how subtyping relationship between individual simple constrained types that take part in two different constrained types leads to a subtyping relationship between the constrained types themselves.

Lemma 5.3 (Independent free variables). Sets of free type variables introduced by different branches of type inference process have an empty intersection. □

Proof. Immediate from the rules in Figure 4.2. □

Lemma 5.4 (Entailment conjunction). If

$$\forall i\colon\quad \mathbb{C} \wedge \{\underline{\alpha}_i \preceq \overline{\alpha}_i\} \vdash \mathrm{glb}_{j \in \underline{J}_i}(\underline{\mathbb{C}}_i^j).\underline{A}_i^j \preceq \mathrm{glb}_{j \in \overline{J}_i}(\overline{\mathbb{C}}_i^j).\overline{A}_i^j \tag{5.19}$$

$$\begin{aligned}\mathbb{C} \wedge \bigwedge_i \{\underline{\beta}_i \preceq \overline{\beta}_i\} \vdash X[\alpha_1/\beta_1', \ldots, \alpha_N/\beta_N', \underline{\beta}_1/\beta_1, \ldots, \underline{\beta}_N/\beta_N]& \\ \preceq X[\alpha_1/\beta_1', \ldots, \alpha_N/\beta_N', \overline{\beta}_1/\beta_1, \ldots, \overline{\beta}_N/\beta_N]&\end{aligned} \tag{5.20}$$

$$FTV(X) = \{\beta_1', \ldots, \beta_N', \beta_1, \ldots, \beta_N\} \tag{5.21}$$

$$FTV(\mathbb{C}) \cap \{\alpha_1, \ldots, \alpha_N, \underline{\alpha}_1, \ldots, \underline{\alpha}_N, \overline{\alpha}_1, \ldots, \overline{\alpha}_N, \underline{\beta}_1, \ldots, \underline{\beta}_N, \overline{\beta}_1, \ldots, \overline{\beta}_N\} = \emptyset \tag{5.22}$$

$$\forall i, i'\colon\quad (\underline{F}_i \cup \{\underline{\alpha}_i\}) \cap (\overline{F}_{i'} \cup \{\overline{\alpha}_{i'}\}) = \emptyset \tag{5.23}$$

$$\forall i, i' \neq i\colon\quad (\underline{F}_i \cup \{\underline{\alpha}_i\}) \cap \underline{F}_{i'} = \emptyset \tag{5.24}$$

$$\forall i, i' \neq i\colon\quad (\overline{F}_i \cup \{\overline{\alpha}_i\}) \cap \overline{F}_{i'} = \emptyset \tag{5.25}$$

where

$$\underline{F}_i \stackrel{\text{def}}{=} \bigcup_{j \in \underline{J}_i} FTV((\underline{\mathbb{C}}_i^j).\underline{A}_i^j) \setminus FTV(\mathbb{C}) \tag{5.26}$$

$$\overline{F}_i \stackrel{\text{def}}{=} \bigcup_{j \in \overline{J}_i} FTV((\overline{\mathbb{C}}_i^j).\overline{A}_i^j) \setminus FTV(\mathbb{C}) \tag{5.27}$$

then

$$\begin{aligned}\mathbb{C} \vdash \mathrm{glb}_{(j_1, \ldots, j_N) \in (\underline{J}_1 \times \cdots \times \underline{J}_N)}(\bigcup_i \underline{\mathbb{C}}_i^{j_i})..X[\underline{\alpha}_1/\beta_1', \ldots, \underline{\alpha}_N/\beta_N', \underline{A}_1^{j_1}/\beta_1, \ldots, \underline{A}_N^{j_N}/\beta_N]& \\ \preceq \mathrm{glb}_{(j_1, \ldots, j_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)}(\bigcup_i \overline{\mathbb{C}}_i^{j_i})..X[\overline{\alpha}_1/\beta_1', \ldots, \overline{\alpha}_N/\beta_N', \overline{A}_1^{j_1}/\beta_1, \ldots, \overline{A}_N^{j_N}/\beta_N]&\end{aligned} \tag{5.28}$$

□

Proof. Let $X = (\mathbb{C}_X).A_X$. Then the conclusion of the lemma is equivalent to

$$\begin{aligned}\bigvee_{(\underline{j}_1, \ldots, \underline{j}_N) \in (\underline{J}_1 \times \cdots \times \underline{J}_N)} \bigwedge_{(\overline{j}_1, \ldots, \overline{j}_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)} (\mathbb{C} \wedge \mathbb{C}_X[\overline{\alpha}_1/\beta_1', \ldots, \overline{\alpha}_N/\beta_N', \overline{A}_1^{\overline{j}_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}_N}/\beta_N]& \\ \wedge (\bigwedge_i \overline{\mathbb{C}}_i^{\overline{j}_i}) \vdash \mathbb{C}_X[\underline{\alpha}_1/\beta_1', \ldots, \underline{\alpha}_N/\beta_N', \underline{A}_1^{\underline{j}_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}_N}/\beta_N] \wedge (\bigwedge_i \underline{\mathbb{C}}_i^{\underline{j}_i})& \\ \wedge A_X[\underline{\alpha}_1/\beta_1', \ldots, \underline{\alpha}_N/\beta_N', \underline{A}_1^{\underline{j}_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}_N}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta_1', \ldots, \overline{\alpha}_N/\beta_N', \overline{A}_1^{\overline{j}_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}_N}/\beta_N])&\end{aligned} \tag{5.29}$$

In order to prove this, it is sufficient to show that

$$\begin{aligned}&\exists(\underline{j}_1', \ldots, \underline{j}_N') \in (\underline{J}_1 \times \cdots \times \underline{J}_N)\colon \\ &\bigwedge_{(\overline{j}_1, \ldots, \overline{j}_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)} (\mathbb{C} \wedge \mathbb{C}_X[\overline{\alpha}_1/\beta_1', \ldots, \overline{\alpha}_N/\beta_N', \overline{A}_1^{\overline{j}_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}_N}/\beta_N] \wedge \bigwedge_i \overline{\mathbb{C}}_i^{\overline{j}_i} \\ &\vdash \mathbb{C}_X[\underline{\alpha}_1/\beta_1', \ldots, \underline{\alpha}_N/\beta_N', \underline{A}_1^{\underline{j}_1'}/\beta_1, \ldots, \underline{A}_N^{\underline{j}_N'}/\beta_N] \wedge \bigwedge_i \underline{\mathbb{C}}_i^{\underline{j}_i'} \\ &\wedge A_X[\underline{\alpha}_1/\beta_1', \ldots, \underline{\alpha}_N/\beta_N', \underline{A}_1^{\underline{j}_1'}/\beta_1, \ldots, \underline{A}_N^{\underline{j}_N'}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta_1', \ldots, \overline{\alpha}_N/\beta_N', \overline{A}_1^{\overline{j}_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}_N}/\beta_N])\end{aligned} \tag{5.30}$$

On the other hand, condition (5.19) is equivalent to

$$\forall i\colon\quad \bigvee_{\underline{j} \in \underline{J}_i} \bigwedge_{\overline{j} \in \overline{J}_i} \mathbb{C} \wedge \{\underline{\alpha}_i \preceq \overline{\alpha}_i\} \wedge \overline{\mathbb{C}}_i^{\overline{j}} \vdash \underline{\mathbb{C}}_i^{\underline{j}} \wedge (\underline{A}_i^{\underline{j}} \preceq \overline{A}_i^{\overline{j}}) \tag{5.31}$$

from which

$$\forall i\ \exists \underline{j}'_i \in \underline{J}_i\colon \quad \bigwedge_{\overline{j}\in\overline{J}_i} \mathbb{C} \wedge \{\underline{\alpha}_i \preceq \overline{\alpha}_i\} \wedge \overline{\mathbb{C}}_i^{\overline{j}} \vdash \underline{\mathbb{C}}_i^{\underline{j}'_i} \wedge (\underline{A}_i^{\underline{j}'_i} \preceq \overline{A}_i^{\overline{j}}) \tag{5.32}$$

Thus

$$\exists(\underline{j}'_1,\ldots,\underline{j}'_N) \in (\underline{J}_1 \times \cdots \times \underline{J}_N)\colon \quad \forall i\colon \quad \bigwedge_{\overline{j}\in\overline{J}_i} \mathbb{C} \wedge \{\underline{\alpha}_i \preceq \overline{\alpha}_i\} \wedge \overline{\mathbb{C}}_i^{\overline{j}} \vdash \underline{\mathbb{C}}_i^{\underline{j}'_i} \wedge (\underline{A}_i^{\underline{j}'_i} \preceq \overline{A}_i^{\overline{j}}) \tag{5.33}$$

In order to prove (5.30), it is sufficient to show that for each $(\overline{j}'_1,\ldots,\overline{j}'_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)$ and any type variable assignment S that satisfies

$$\mathbb{C} \wedge \mathbb{C}_X[\overline{\alpha}_1/\beta'_1,\ldots,\overline{\alpha}_N/\beta'_N,\overline{A}_1^{\overline{j}'_1}/\beta_1,\ldots,\overline{A}_N^{\overline{j}'_N}/\beta_N] \wedge \bigwedge_i \overline{\mathbb{C}}_i^{\overline{j}'_i} \tag{5.34}$$

there exists an extension S_e that satisfies

$$\begin{aligned}&\mathbb{C}_X[\underline{\alpha}_1/\beta'_1,\ldots,\underline{\alpha}_N/\beta'_N,\underline{A}_1^{\underline{j}'_1}/\beta_1,\ldots,\underline{A}_N^{\underline{j}'_N}/\beta_N] \wedge \bigwedge_i \underline{\mathbb{C}}_i^{\underline{j}'_i}\\ &\wedge A_X[\underline{\alpha}_1/\beta'_1,\ldots,\underline{\alpha}_N/\beta'_N,\underline{A}_1^{\underline{j}'_1}/\beta_1,\ldots,\underline{A}_N^{\underline{j}'_N}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta'_1,\ldots,\overline{\alpha}_N/\beta'_N,\overline{A}_1^{\overline{j}'_1}/\beta_1,\ldots,\overline{A}_N^{\overline{j}'_N}/\beta_N]\end{aligned} \tag{5.35}$$

Let $(\overline{j}'_1,\ldots,\overline{j}'_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)$ be arbitrary indices and S an arbitrary type assignment that satisfies (5.34). If no such S exists, the statement is trivially true. Define

$$S'_i(\gamma) = \begin{cases} S(\overline{\alpha}_i) & \text{if } \gamma \equiv \underline{\alpha}_i \\ S(\gamma) & \text{otherwise}\end{cases} \tag{5.36}$$

S'_i is an extension of S since

$$\underline{\alpha}_i \notin \left(FTV(\mathbb{C}) \cup \bigcup_{\overline{j}\in\overline{J}_i} (FTV(\overline{A}_i^{\overline{j}}) \cup FTV(\overline{\mathbb{C}}_i^{\overline{j}})\right)$$

from conditions (5.22) and (5.23) of the lemma. Furthermore, S'_i satisfies

$$\mathbb{C} \wedge \overline{\mathbb{C}}_i^{\overline{j}'_i} \wedge (\underline{\alpha}_i \preceq \overline{\alpha}_i) \tag{5.37}$$

for all i (from (5.34) and (5.36)). From this and (5.33) there exists an extension S''_i of S'_i that satisfies

$$\underline{\mathbb{C}}_i^{\underline{j}'_i} \wedge (\underline{A}_i^{\underline{j}'_i} \preceq \overline{A}_i^{\overline{j}}) \tag{5.38}$$

Let

$$S''(\gamma) = \left\{ S''_i(\gamma) \quad \text{if } \gamma \in \mathrm{dom}(S''_i) \right. \tag{5.39}$$

This definition is valid since

$$\forall i, i' \neq i\colon \quad \mathrm{dom}(S''_i) \cap \mathrm{dom}(S''_{i'}) = \mathrm{dom}(S)$$

(from conditions (5.24) and (5.25) of the lemma) and

$$\forall \gamma \in \mathrm{dom}(S), i, i'\colon \quad S''_i(\gamma) = S''_{i'}(\gamma) = S(\gamma)$$

since S_i'' are extensions of S. S'' thus defined is an extension of S and S_i' for each i. It also satisfies (5.38) by definition of S'' and S_i''.

Let $\underline{\beta}_i = \underline{A}_i^{\underline{j}'_i}$, $\overline{\beta}_i = \overline{A}_i^{\overline{j}'_i}$, and $\alpha_i = \overline{\alpha}_i$. Then condition (5.20) implies that

$$\begin{aligned}
&\mathbb{C} \wedge \bigwedge_i \{\underline{A}_i^{\underline{j}'_i} \preceq \overline{A}_i^{\overline{j}'_i}\} \wedge C_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \overline{A}_1^{\overline{j}'_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}'_N}/\beta_N] \vdash \\
&\qquad C_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \wedge \\
&(A_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \overline{A}_1^{\overline{j}'_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}'_N}/\beta_N])
\end{aligned} \tag{5.40}$$

The variable assignment S'' satisfies

$$\mathbb{C} \wedge \bigwedge_i \{\underline{A}_i^{\underline{j}'_i} \preceq \overline{A}_i^{\overline{j}'_i}\} \wedge C_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \overline{A}_1^{\overline{j}'_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}'_N}/\beta_N]$$

(by (5.34) and (5.38)) and therefore there exists an extension S_e of S'' such that S_e satisfies

$$\begin{aligned}
&C_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \wedge \\
&(A_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \overline{A}_1^{\overline{j}'_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}'_N}/\beta_N])
\end{aligned} \tag{5.41}$$

(it follows from (5.40)). However,

$$\forall i: \quad S_e(\underline{\alpha}_i) = S_e(\overline{\alpha}_i)$$

(by definition of S_i' and from the fact that S_e is an extension of S_i' for all i). Therefore, since S_e satisfies (5.41), it also satisfies

$$\begin{aligned}
&C_X[\underline{\alpha}_1/\beta'_1, \ldots, \underline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \wedge \\
&(A_X[\underline{\alpha}_1/\beta'_1, \ldots, \underline{\alpha}_N/\beta'_N, \underline{A}_1^{\underline{j}'_1}/\beta_1, \ldots, \underline{A}_N^{\underline{j}'_N}/\beta_N] \preceq A_X[\overline{\alpha}_1/\beta'_1, \ldots, \overline{\alpha}_N/\beta'_N, \overline{A}_1^{\overline{j}'_1}/\beta_1, \ldots, \overline{A}_N^{\overline{j}'_N}/\beta_N])
\end{aligned} \tag{5.42}$$

From this and (5.38) S_e satisfies (5.35).

Thus, it has been shown that for any $(\overline{j}'_1, \ldots, \overline{j}'_N) \in (\overline{J}_1 \times \cdots \times \overline{J}_N)$ and any type variable assignment S that satisfies (5.34) there exists an extension S_e that satisfies (5.35) if the conditions of the lemma are satisfied. This proves the statement of the lemma. □

The following defines subtyping relationship between typing environments as a straightforward extension of subtyping relationship between types.

Definition 5.15 (Typing environment ordering). If $\overline{\Theta}$ and $\underline{\Theta}$ are typing environments, then

$$(\mathbb{C} \vdash \underline{\Theta} \preceq \overline{\Theta}) \overset{\text{def}}{\Longleftrightarrow} (\forall \langle \mathbf{name} \rangle \in Id: \quad \mathbb{C} \vdash \underline{\Theta}(\langle \mathbf{name} \rangle) \preceq \overline{\Theta}(\langle \mathbf{name} \rangle))$$

□

Theorem 5.7 (Static subsumption). If

$$\overline{\Theta} \rhd \langle \mathbf{expr} \rangle : \overline{X} \tag{5.43}$$

$$\mathbb{C} \vdash \underline{\Theta} \preceq \overline{\Theta} \tag{5.44}$$

then

$$\underline{\Theta} \triangleright \langle\mathbf{expr}\rangle : \underline{X} \tag{5.45}$$
$$\mathbb{C} \vdash \underline{X} \preceq \overline{X} \tag{5.46}$$

□

Proof. The proof is done by transforming the derivation of (5.43) into the derivation of (5.45). This is done by induction on the depth of the derivation tree. Since the expression is the same in both cases, it is possible to apply the same rules in the same order to obtain the necessary derivation.

Base: The derivation of (5.43) has depth 1. Thus the derivation includes only one application of the rule Axiom, the expression $\langle\mathbf{expr}\rangle$ consists of a single variable $\mathbf{u}$, and $\overline{\Theta}(\mathbf{u}) = \overline{X}$. Then, $\underline{\Theta}(\mathbf{u}) = \underline{X}$ and $\mathbb{C} \vdash \underline{X} \preceq \overline{X}$ since $\mathbb{C} \vdash \underline{\Theta} \preceq \overline{\Theta}$. Thus, it is possible to build the derivation of (5.45) as a single application of the rule Axiom, where the derived type is $\underline{X}$. Since $\mathbb{C} \vdash \underline{X} \preceq \overline{X}$, the base case is proven.

Induction step: Assume that for all derivations of (5.43) of depth less or equal to $n-1$ $(n > 1)$ the statement of the theorem is proven. It will be shown that in this case the statement of the theorem is also true for all derivations of depth n. The proof is by cases depending on the first (root) rule applied in the derivation of (5.43).

The general schema is as follows: first, it is shown that the typing environments in the premises of each rule have the same relationships as the ones in the rule conclusion. Then the induction assumption is used to establish the subtyping relationship between types inferred for the premises of the rule. Finally, Lemma 5.4 is used to derive the subtyping relationship between the types derived in the conclusion of the rule. In all cases, conditions (5.22), (5.25), (5.24), and (5.23) of Lemma 5.4 follow from Lemma 5.3 and the fact that new variables introduced in the derivation are always fresh. Condition (5.19) is always a consequence of the induction hypothesis. Condition (5.21) is satisfied by the appropriate choice of X. The only condition that has to be proven individually for each case is the condition (5.20). The set of indices i for each rule is the set of numbers from 1 to N, where N is the number of premises in the rule under consideration.

Rule Axiom: Can't happen, since if the root rule is Axiom, then $n = 1$.

Rule Abs: The detailed proof will be given for this case only, since all other cases are similar and differ primarily in definition of X used in Lemma 5.4.

Since this rule is the root one, the expression under consideration is **fun (x)** $\langle\mathbf{expr}\rangle$.

First, it will be shown that if $\mathbb{C} \vdash \underline{\Theta} \preceq \overline{\Theta}$, then

$$\mathbb{C} \uplus \{\underline{\alpha} \preceq \overline{\alpha}\} \vdash (\underline{\Theta}' = \underline{\Theta} \uplus \{\mathbf{x} : \underline{\alpha}\} \preceq \overline{\Theta} \uplus \{\mathbf{x} : \overline{\alpha}\} = \overline{\Theta}') \tag{5.47}$$

where $\underline{\alpha}, \overline{\alpha}$ are fresh. Since $\underline{\Theta}'(\mathbf{y}) = \underline{\Theta}(\mathbf{y})$ and $\overline{\Theta}'(\mathbf{y}) = \overline{\Theta}(\mathbf{y})$ for all $\mathbf{y} \neq \mathbf{x}$, it is sufficient to show that $\mathbb{C} \uplus \{\underline{\alpha} \preceq \overline{\alpha}\} \vdash (\underline{\alpha} = \underline{\Theta}'(\mathbf{x}) \preceq \overline{\Theta}'(\mathbf{x}) = \overline{\alpha})$, i.e. $\mathbb{C} \uplus \{\underline{\alpha} \preceq \overline{\alpha}\} \vdash \underline{\alpha} \preceq \overline{\alpha}$, which is always true as $\{\underline{\alpha}, \overline{\alpha}\} \cap FTV(\mathbb{C}) = \emptyset$. Thus, equation (5.47) has been proven.

Since the rule **Abs** has been successfully used, its conditions are satisfied and therefore

$$\overline{\Theta}' \triangleright \langle\mathbf{expr}\rangle : \overline{X}' \tag{5.48}$$

where the derivation of equation (5.48) has depth $n - 1$. By induction assumption and from equation (5.47)

$$\underline{\Theta}' \triangleright \langle\mathbf{expr}\rangle : \underline{X}' \tag{5.49}$$

and

$$\mathbb{C} \uplus \{\underline{\alpha} \preceq \overline{\alpha}\} \vdash \underline{X}' \preceq \overline{X}' \tag{5.50}$$

Therefore,

$$\underline{\Theta} \triangleright \texttt{fun (x) } \langle\texttt{expr}\rangle : \underline{X} \tag{5.51}$$

where

$$\underline{X} = \mathrm{glb}_i((\underline{\mathbb{C}}^i).\underline{\alpha} \rightarrow \underline{A}^i) \tag{5.52}$$

$$\underline{X}' = \mathrm{glb}_i((\underline{\mathbb{C}}^i).\underline{A}^i) \tag{5.53}$$

From the original derivation, it is also known that

$$\overline{\Theta} \triangleright \texttt{fun (x) } \langle\texttt{expr}\rangle : \overline{X} \tag{5.54}$$

$$\overline{X} = \mathrm{glb}_i((\overline{\mathbb{C}}^i).\overline{\alpha} \rightarrow \overline{A}^i) \tag{5.55}$$

$$\overline{X}' = \mathrm{glb}_i((\overline{\mathbb{C}}^i).\overline{A}^i) \tag{5.56}$$

The only thing left to prove is therefore

$$\mathbb{C} \vdash \underline{X} \preceq \overline{X} \tag{5.57}$$

This can be obtained by using Lemma 5.4 with $N = 1$ and

$$X[\alpha, \beta] \stackrel{\mathrm{def}}{=} ().\alpha \rightarrow \beta$$

In order to use Lemma 5.4, it has to be shown that condition (5.20) is satisfied, i.e.

$$\mathbb{C} \wedge (\underline{\beta} \preceq \overline{\beta}) \vdash X[\alpha/\alpha, \underline{\beta}/\beta] \preceq X[\alpha/\alpha, \overline{\beta}/\beta]$$

This follows from the fact that

$$\begin{aligned} &(\underline{\beta} \preceq \overline{\beta}) \vdash \alpha \rightarrow \underline{\beta} \preceq \alpha \rightarrow \overline{\beta} \\ &\equiv (\underline{\beta} \preceq \overline{\beta}) \vdash (\alpha \preceq \alpha) \wedge (\underline{\beta} \preceq \overline{\beta}) \end{aligned}$$

which is immediately true.

Rule Appl: Here $N = 2$ and X is chosen to be

$$X[\alpha_1, \alpha_2, \beta_1, \beta_2] \stackrel{\mathrm{def}}{=} (\beta_1 \preceq \beta_2 \rightarrow \alpha_1).\alpha_1$$

Condition (5.20) necessary for the applicability of Lemma 5.4 follows from

$$\begin{aligned} &(\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \vdash (\underline{\beta}_1 \preceq \underline{\beta}_2 \rightarrow \alpha_1).\alpha_1 \preceq (\overline{\beta}_1 \preceq \overline{\beta}_2 \rightarrow \alpha_1).\alpha_1 \\ &\equiv (\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \wedge (\overline{\beta}_1 \preceq \overline{\beta}_2 \rightarrow \alpha_1) \vdash (\underline{\beta}_1 \preceq \underline{\beta}_2 \rightarrow \alpha_1) \wedge (\alpha_1 \preceq \alpha_1) \\ &\equiv (\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \wedge (\overline{\beta}_1 \preceq \overline{\beta}_2 \rightarrow \alpha_1) \vdash (\underline{\beta}_1 \preceq \underline{\beta}_2 \rightarrow \alpha_1) \end{aligned}$$

which is immediately true as

$$\underline{\beta}_1 \preceq \overline{\beta}_1 \preceq \overline{\beta}_2 \rightarrow \alpha_1 \preceq \underline{\beta}_2 \rightarrow \alpha_1$$

Rule Product: Here $N = n$ and X is chosen to be

$$X[\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_n] \stackrel{\mathrm{def}}{=} ().(\beta_1, \ldots, \beta_n)$$

Condition (5.20) necessary for the applicability of Lemma 5.4 follows from

$$\begin{aligned} &\bigwedge_i (\underline{\beta}_i \preceq \overline{\beta}_i) \vdash (\underline{\beta}_1, \ldots, \underline{\beta}_n) \preceq (\overline{\beta}_1, \ldots, \overline{\beta}_n) \\ &\equiv \bigwedge_i (\underline{\beta}_i \preceq \overline{\beta}_i) \vdash \bigwedge_i (\underline{\beta}_i \preceq \overline{\beta}_i) \end{aligned}$$

which is immediately true.

Rule Seq: Here $N = 2$ and X is chosen to be

$$X[\alpha_1, \alpha_2, \beta_1, \beta_2] \overset{\text{def}}{=} ().\beta_2$$

Condition (5.20) necessary for the applicability of Lemma 5.4 follows from

$$(\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \vdash \underline{\beta}_2 \preceq \overline{\beta}_2$$

which is immediately true.

Rule Let: Before Lemma 5.4 can be used here, it has to be shown that

$$\underline{\Theta} \preceq \overline{\Theta} \Rightarrow \underline{\Theta} \uplus \{\mathbf{x} : \underline{X}_1\} \preceq \overline{\Theta} \uplus \{\mathbf{x} : \overline{X}_1\} \tag{5.58}$$

where

$$\begin{aligned} &\underline{\Theta} \rhd \langle\mathbf{expr}\rangle_1 : \underline{X}_1 \\ &\overline{\Theta} \rhd \langle\mathbf{expr}\rangle_1 : \overline{X}_1 \end{aligned} \tag{5.59}$$

However, if $\underline{\Theta} \preceq \overline{\Theta}$, then

$$\underline{\Theta} \uplus \{\mathbf{x} : \underline{X}_1\} \preceq \overline{\Theta} \uplus \{\mathbf{x} : \overline{X}_1\}$$

follows from equations (5.59), induction hypothesis, and the definition of $\uplus$. Thus (5.58) is proven and it is now possible to use Lemma 5.4. Let

$$X[\alpha_1, \alpha_2, \beta_1, \beta_2] \overset{\text{def}}{=} ().\beta_2$$

Then condition (5.20) necessary for the applicability of Lemma 5.4 follows from

$$(\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \vdash \underline{\beta}_2 \preceq \overline{\beta}_2$$

which is immediately true.

Rule TypedLet: Before Lemma 5.4 can be used here, it has to be shown that

$$\underline{\Theta} \preceq \overline{\Theta} \Rightarrow \underline{\Theta} \uplus \{\mathbf{x} : T'\} \preceq \overline{\Theta} \uplus \{\mathbf{x} : T'\} \tag{5.60}$$

where $T' \overset{\text{def}}{=} expand_{\mathcal{G}}(T)$. However, if $\underline{\Theta} \preceq \overline{\Theta}$, then

$$\underline{\Theta} \uplus \{\mathbf{x} : T'\} \preceq \overline{\Theta} \uplus \{\mathbf{x} : T'\}$$

follows from the definition of $\uplus$. Thus (5.60) is proven and it is now possible to use Lemma 5.4. Let

$$X[\alpha_1, \alpha_2, \beta_1, \beta_2] \overset{\text{def}}{=} (\beta_1 \preceq T').\beta_2$$

Then condition (5.20) necessary for the applicability of Lemma 5.4 follows from

$$\begin{aligned} (\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) &\vdash (\underline{\beta}_1 \preceq T').\underline{\beta}_2 \preceq (\overline{\beta}_1 \preceq T').\overline{\beta}_2 \\ &\equiv (\underline{\beta}_1 \preceq \overline{\beta}_1) \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \wedge (\overline{\beta}_1 \preceq T') \vdash (\underline{\beta}_1 \preceq T') \wedge (\underline{\beta}_2 \preceq \overline{\beta}_2) \end{aligned}$$

which is immediately true as

$$\underline{\beta}_1 \preceq \overline{\beta}_1 \preceq T'$$

Thus, for all cases it has been shown that the induction assumption for all $n < N$ where $N > 1$ implies the statement of the theorem for N. The induction base (for $N = 1$) has also been proven, and therefore the statement of the theorem is true for all N. □

5.3.2 Natural semantics

This section defines the natural semantics of the target language that will be used to show the soundness of the presented type-checking algorithm.

The run-time objects (*rt-objects*) of the language comprise the set $\mathcal{O}$. Rt-objects will be denoted as **a**, **b**, **c**, ... Each rt-object of the language has a type. This will be written as **object** : **T_Type**. Rt-objects of the language are:

Atomic constants These are constants explicitly defined by the program and the initial environment, such as `5`, **unit** etc. Behaviors declared in the program are considered to be atomic constants as well. The set $\mathcal{B} \subseteq \mathcal{O}$ is the set of all behaviors declared by the program.

Products A product object is an ordered list of run-time objects:

$$(\mathbf{a}_1, \ldots, \mathbf{a}_n)$$

where $\mathbf{a}_i \in \mathcal{P}(\mathcal{O})$. Products of rt-objects comprise the set $\mathcal{P}(\mathcal{O})$. As usual, it is assumed that

$$(\mathbf{a}) \equiv \mathbf{a}$$

and therefore $\mathcal{O} \subseteq \mathcal{P}(\mathcal{O})$.

Closures Closures are function abstractions together with the environment they are defined in. They are denoted as $closure(E, \mathbf{x}, \mathbf{expr})$. Here E is the environment, **x** is the formal argument, and **expr** is the body of the abstraction.

Run-time objects are irreducible and can appear as the final result of a successful program execution (reduction).

The *run-time environment* (*rt-environment*) E is a set of name-to-rt-object bindings of the form $\langle\mathbf{name}\rangle = \mathbf{a}$, where $\mathbf{name} \in Id$ (set of identifiers) and $\mathbf{a} \in \mathcal{O}$. Given an rt-environment E, $E(\langle\mathbf{name}\rangle)$ denotes the rt-object **a** in the binding of the form $\langle\mathbf{name}\rangle = \mathbf{a}$ in E or $\emptyset$ if $\langle\mathbf{name}\rangle$ is unbound in E. The operation $\uplus$: $E_1 \uplus E_2$ constructs the new rt-environment that includes all bindings in E_2 and those bindings of E_1 that are not overridden in E_2. Formally,

$$(E_1 \uplus E_2)(\langle\mathbf{name}\rangle) = \begin{cases} E_2(\langle\mathbf{name}\rangle) & \text{if } E_2(\langle\mathbf{name}\rangle) \neq \emptyset \\ E_1(\langle\mathbf{name}\rangle) & \text{if } E_2(\langle\mathbf{name}\rangle) = \emptyset \end{cases}$$

The *store* S is an unspecified entity that represents the storage space of the program. For the purposes of this section, it is sufficient to state that there exists a predicate $Valid_\Theta(S)$ defined on all possible stores; an initial store $\hat{S}$ such that $Valid_{\Theta_0}(\hat{S}) = TRUE$; and a set of *primitive functions* $\{primitive_i\}$. The validity of a store may or may not depend on the typing environment Θ.

Primitive functions are introduced to model low-level system primitives. Each primitive function is a partial function that maps pairs $(S, \mathbf{a})$ to $(S', \mathbf{a}')$. This mapping has the following meaning: if $primitive_i(S, \mathbf{a}) = (S', \mathbf{a}')$, then the primitive function $primitive_i$ applied to an rt-object **a** in the presence of the store S produces the result $\mathbf{a}'$ and changes the store to become S'.

Primitive functions are the most basic execution primitives and correspond to low-level implementation functions. Their correctness is not checked; rather, it is assumed. The right set of primitive functions along with their definitions and the proof of their correctness is the responsibility of the language designer.

Before the natural semantics can be defined, certain assumptions about the underlying user type graph $\mathcal{G}$ need to be established. It is assumed that $\mathcal{G}$ has the following properties:

Products: It is assumed that the following definitions are given:

```
concrete type T_ProductN(covar X1, ..., covar XN);
```

for all **N** from 2 to M, where M is the maximum arity of product expressions in the program (M is finite since every program is a finite text; at the same time, products can not grow dynamically in the proposed semantics). Existence of the type definition

```
concrete type T_Unit;
```

is also assumed. The type **T_Unit** can be thought of as a 0-ary product type. It is further assumed that none of the above product types has any subtypes or concrete supertypes. The existence of a predefined rt-object **unit** of type **T_Unit** is also assumed.

Functionals: It is assumed that behavior and function types are defined as

```
concrete type T_Function(contravar A, covar R);
concrete type T_Behavior(contravar A, covar R)
                                   subtype of T_Function(A, R);
```

It is assumed that there are no concrete subtypes of **T_Behavior** and that the only concrete subtype of **T_Function** is **T_Behavior**. Instead of **T_Function**(A, R) and **T_Behavior**(A, R) the denotations $A \rightarrow R$ and $A \rightarrow_b R$, respectively, will be used.

The assumptions about the product types play a crucial role in definitions of behavior consistency and the treatment and typing of product expressions. Assumptions about functional types are used in typing and treatment of abstraction and application expressions. The assumptions about non-existence of concrete subtypes of the functional types can be lifted provided the appropriate reduction rule for application expression is added.

The *natural semantics* of the target language is defined in Figure 5.1. The statement of the form

$$(S, E, \langle \mathbf{expr} \rangle) \Downarrow_{\mathcal{G}} (S', \mathsf{a})$$

means that the expression $\langle \mathbf{expr} \rangle$ evaluated in the environment E with the store S reduces to rt-object a and changes the store to become S'.

The function *dispatch*(b, a) ($\mathcal{B} \times \mathcal{O} \rightarrow \mathcal{O}$) is defined by Definition 4.21.

In this section, natural semantics of the target language has been defined. The next section discusses issues related to run-time object typing.

5.3.3 Execution state typing

In this section, typing of run-time objects and correctness of a particular state of program execution will be defined. The correctness condition placed upon primitive functions and their associations will also be formulated.

Definition 5.16 (Extended typing environment). An *extended typing environment* Θ is a set of pairs $a : X$ where X is a type and $a \in (Id \cup \mathcal{O})$. □

An extended typing environment is a straightforward extension of typing environment designed to handle typing of run-time objects. In this section, the term "typing environment" will be used to mean "extended typing environment". The notion of *initial extended typing environment* Θ_0 is defined as a straightforward extension of initial typing environment (Definition 4.11).

Definition 5.17 (Run-time object typing). If Θ is a typing environment (Definition 5.16) and a is an rt-object, then its typing is defined by the rules depicted in Figure 5.2 which augment the rules in Figure 4.2. Here $\Theta(\Theta, E)$ denotes the environment obtained as follows:

$$\Theta(\Theta, E)(\langle \mathbf{name} \rangle) = \begin{cases} X & \text{if } E(\langle \mathbf{name} \rangle) \neq \emptyset \text{ and } \Theta \triangleright E(\langle \mathbf{name} \rangle) : X \\ \top & \text{otherwise} \end{cases}$$

□

Definition 5.18 (RT-similarity). Two types A_1 and A_2 are called *rt-similar* iff one of the following is true

$$\frac{E(\langle\mathbf{name}\rangle) \neq \emptyset}{(S, E, \langle\mathbf{name}\rangle) \Downarrow_{\mathcal{G}} (S, E(\langle\mathbf{name}\rangle))}\ \textsf{RedAxiom}$$

$$\frac{}{(S, E, \mathtt{fun}\ \ (\mathtt{x})\ \ \langle\mathbf{expr}\rangle) \Downarrow_{\mathcal{G}} (S, \mathit{closure}(E, \mathtt{x}, \langle\mathbf{expr}\rangle))}\ \textsf{RedFunction}$$

$$\frac{(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathrm{a}_1) \qquad \ldots \qquad (S_{n-1}, E, \langle\mathbf{expr}\rangle_n) \Downarrow_{\mathcal{G}} (S_n, \mathrm{a}_n)}{(S_0, E, (\langle\mathbf{expr}\rangle_1, \ldots, \langle\mathbf{expr}\rangle_n)) \Downarrow_{\mathcal{G}} (S_n, (\mathrm{a}_1, \ldots, \mathrm{a}_n))}\ \textsf{RedProduct}$$

$$\frac{(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathrm{a}_1) \qquad (S_1, E \uplus \{\langle\mathbf{name}\rangle = \mathrm{a}_1\}, \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2)}{(S_0, E, \mathtt{let}\ [T]\ \langle\mathbf{name}\rangle\mathtt{=}\langle\mathbf{expr}\rangle_1\ \ \mathtt{in}\ \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2)}\ \textsf{RedLet}$$

$$\frac{(S_0, E, \langle\mathbf{exprs}\rangle) \Downarrow_{\mathcal{G}} (S_1, \mathrm{a}_1) \qquad (S_1, E, \langle\mathbf{expr}\rangle) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2)}{(S_0, E, \langle\mathbf{exprs}\rangle\,\mathtt{;}\langle\mathbf{expr}\rangle) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2)}\ \textsf{RedSequence}$$

$$\frac{\begin{array}{c}(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathit{closure}(E', \mathtt{x}, \langle\mathbf{expr}\rangle)) \qquad (S_1, E, \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2) \\ (S_2, E \uplus E' \uplus \{\mathtt{x} = \mathrm{a}_2\}, \langle\mathbf{expr}\rangle) \Downarrow_{\mathcal{G}} (S_3, \mathrm{a}_3)\end{array}}{(S_0, E, \langle\mathbf{expr}\rangle_1 \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_3, \mathrm{a}_3)}\ \textsf{RedFuncApplication}$$

$$\frac{\begin{array}{c}(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathtt{behavior}) \qquad \mathtt{behavior} \in \mathcal{B} \\ (S_1, E, \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2) \\ \mathit{dispatch}(\mathtt{behavior}, \mathrm{a}_2) = \mathtt{fun}\ \ (\mathtt{x})\ \ \langle\mathbf{expr}\rangle \\ (S_2, E, (\ \mathtt{fun}\ \ (\mathtt{x})\ \ \langle\mathbf{expr}\rangle\ \ )\ \mathrm{a}) \Downarrow_{\mathcal{G}} (S_3, \mathrm{a}_3)\end{array}}{(S_0, E, \langle\mathbf{expr}\rangle_1 \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_3, \mathrm{a}_3)}\ \textsf{RedBehApplication}$$

$$\frac{\begin{array}{c}(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathtt{behavior}) \qquad \mathtt{behavior} \in \mathcal{B} \\ (S_1, E, \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_2, \mathrm{a}_2) \\ \mathit{dispatch}(\mathtt{behavior}, \mathrm{a}_2) = \mathit{primitive}_i \\ \mathit{primitive}_i(S_2, \mathrm{a}_2) = (S_3, \mathrm{a}_3)\end{array}}{(S_0, E, \langle\mathbf{expr}\rangle_1 \langle\mathbf{expr}\rangle_2) \Downarrow_{\mathcal{G}} (S_3, \mathrm{a}_3)}\ \textsf{RedBehPrimApplication}$$

Figure 5.1: Natural semantics of the target language

$$\frac{\Theta(\mathsf{a}) = X}{\Theta \triangleright \mathbf{a} : X}\ \textsf{RTAxiom}$$

$$\frac{\Theta \triangleright \mathbf{a}_1 : \mathrm{glb}_{j_1}((\mathbb{C}_1^{j_1}).A_1^{j_1}) \qquad \ldots \qquad \Theta \triangleright \mathbf{a}_n : \mathrm{glb}_{j_n}((\mathbb{C}_n^{j_n}).A_n^{j_n})}{\Theta \triangleright (\mathbf{a}_1, \ldots, \mathbf{a}_n) : \mathrm{glb}_{j_1,\ldots,j_n}((\cup_i \mathbb{C}_i^{j_i}).(A_1^{j_1}, \ldots, A_n^{j_n}))}\ \textsf{RTProduct}$$

$$\frac{\Theta \uplus \Theta(\Theta, E) \triangleright \mathtt{fun}\ \ (\mathtt{x})\ \ \langle\mathbf{expr}\rangle : X}{\Theta \triangleright \mathit{closure}(E, \mathtt{x}, \langle\mathbf{expr}\rangle) : X}\ \textsf{RTClosure}$$

Figure 5.2: Typing rules for rt-objects

1. $A_1 = A_2$
2. $A_1 = a(A_1^1, \ldots, A_1^{Arity(a)})$, $A_2 = a(A_2^1, \ldots, A_2^{Arity(a)})$, and for each i A_1^i is rt-similar to A_2^i
3. $A_1 = a(A_1^1, \ldots, A_1^{Arity(a)})$, $A_2 = a(A_2^1, \ldots, A_2^{Arity(a)})$, and $a \in \{\rightarrow, \rightarrow_b\}$

□

Rt-similarity requires that two types be identical when their leaves related to functional types are cut off. For example, $(a, (b \rightarrow c, d), e)$ is rt-similar to $(a, (f \rightarrow g, d), e)$, but $(a, (b, c, d), e)$ is not rt-similar to $(a, (f, g, d), e)$.

Definition 5.19 (Run-time pre-types). *Run-time pre-type* p over $\mathcal{G}$ is defined recursively as follows:

1. If $p = A$ where A is a type over $\mathcal{G} \cup FTV$ such that $pfunctor(A) \in Concrete_{\mathcal{G}}$ and
$$pfunctor(A) \notin \{\rightarrow, \rightarrow_b\} \Rightarrow FTV(A) = \emptyset$$
then p is a run-time pre-type.
2. If $p = (p_1, \ldots, p_n)$ where p_i are run-time pre-types such that
$$\forall i, j \neq i: \quad FTV(p_i) \cap FTV(p_j) = \emptyset$$
then p is a run-time pre-type.

□

Thus a run-time pre-type can contain variables in the leaves of functional types only. Thus $(a(b), \alpha \rightarrow c)$ is a run-time pre-type while $(a(b), c(\alpha))$ is not.

Definition 5.20 (Run-time types). *Run-time type* q over $\mathcal{G}$ is defined as follows:

$$q = \mathrm{glb}_i(\mathbb{C}^i).p_i$$

is a run-time type iff the following holds:

$$\forall i: \quad p_i \text{ is a run-time pre-type} \tag{5.61}$$

$$\forall i: \quad \vdash_{\mathcal{G}} \mathbb{C}^i \tag{5.62}$$

$$\forall i, j: \quad p_i \text{ and } p_j \text{ are RT-similar} \tag{5.63}$$

The set of all run-time types is denoted $RTyp_{\mathcal{G}}$. □

Run-time types are thus greatest lower bounds of a finite number of rt-similar consistent run-time pre-types. For example,

$$\mathrm{glb}((\alpha \preceq a).\alpha \rightarrow \beta, a \rightarrow b) \tag{5.64}$$

is a run-time type while

$$\mathrm{glb}((\alpha \preceq a).\alpha \rightarrow \beta, a)$$

is not.

Definition 4.15 (Primary form of run-time types; extended from page 132 above). $primary_{\mathcal{G}}$ is extended to work for run-time types in the following manner:

$$primary_{\mathcal{G}}(q) \overset{\mathrm{def}}{=} \{\langle c_1, \ldots, c_n \rangle \mid c_j = pfunctor(q_j^i) \text{ for some } i\}$$

where

$$q = \mathrm{glb}_i(\mathbb{C}^i).(q_1^i, \ldots, q_n^i)$$

□

This definition is valid since *pfunctor*(q_j^i) is independent of i according to (5.63). For example, the primary form of the type (5.64) is $\langle \rightarrow \rangle$.

Having defined the run-time object typing, it is now possible to formally specify the requirements placed on primitive functions. The primitive functions are assumed to be *correct* in the following sense:

Definition 5.21 (Primitive function correctness). A primitive function *primitive*$_i$ is *correct* w.r.t Θ iff for every behavior association of the form

$$\textbf{behavior}\ (\mathbb{C})\ A \rightarrow R\ (\textbf{name}) : \mathit{primitive}_i$$

for any store S such that $\mathit{Valid}_\Theta(S) = \mathit{TRUE}$, for each rt-object $\mathbf{a}$ such that $\Theta \triangleright \mathbf{a} : X^a$, $X^a = \mathrm{glb}_i(\mathbb{C}_i^a).A_i$, and

$$\exists i\colon\quad \vdash_{\mathcal{G}} \mathbb{C}_i^a \wedge \mathbb{C} \wedge A_i \preceq A$$

the following holds:

$$\begin{aligned} \mathit{primitive}_i(S, \mathbf{a}) &= (S', \mathbf{r}) \\ \mathit{Valid}_\Theta(S') &= \mathit{TRUE} \\ \Theta \triangleright \mathbf{r} : X^r &= \mathrm{glb}_j(\mathbb{C}_j^r).R_j \\ \forall i \exists j\colon\quad \mathbb{C} \wedge \mathbb{C}_i^a \wedge (A_i \preceq A) &\vdash_{\mathcal{G}} \mathbb{C}_j^r \wedge (R_j \preceq R) \end{aligned}$$

□

In other words, the primitive functions are required to behave just as good as type-checked non-primitive ones w.r.t their type specifications. Note that there is no definition of store consistency here as it depends upon the semantics of primitive functions. However, the results of this chapter do not depend on the definition of store consistency. It is only required that correct applications of primitive functions do not disturb it (as stated in the above definition). An example of a set of primitive functions satisfying these conditions is given in Section 5.4.

In this section, execution state (run-time object) typing was defined. Next section proves a set of correctness properties for behavior dispatch process.

5.3.4 Dispatch correctness

In this section, several properties of *dispatch* function used in Figure 5.1 are proven. The ultimate goal is to prove that in a correctly typechecked expression the dispatch is always valid and correct (no "message not understood" or "message ambiguous" errors).

Lemma 5.5 (Argument property). If $q \in \mathit{RTyp}_{\mathcal{G}}$ then

$$(\exists i\colon\quad \vdash_{\mathcal{G}} \mathbb{C} \wedge \mathbb{C}_i^q \wedge (A_i^q \preceq A)) \Rightarrow \mathit{primary}_{\mathcal{G}}(q) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A) \tag{5.65}$$

$$\forall i\colon\quad \mathbb{C} \wedge \mathbb{C}_i^q \wedge (A_i^q \preceq A) \vdash_{\mathcal{G}} \mathbb{C}^{qp} \wedge A_i^q \preceq A^{qp} \preceq A \tag{5.66}$$

$$\forall i\colon\quad \exists t \in \mathit{concrete}_{\mathcal{G}}((\mathbb{C}).A)\colon\quad \mathbb{C} \wedge \mathbb{C}_i^q \wedge (A_i^q \preceq A) \vdash_{\mathcal{G}} \mathbb{C}^t \wedge A_i^q \preceq A^t \preceq A \tag{5.67}$$

where $q = \mathrm{glb}_i(\mathbb{C}_i^q).A_i^q$, $\mathit{abstract}_{\mathcal{G}}(\mathit{primary}_{\mathcal{G}}(t)) = (\mathbb{C}^t).A^t$ and $\mathit{abstract}_{\mathcal{G}}(\mathit{primary}_{\mathcal{G}}(q)) = (\mathbb{C}^{qp}).A^{qp}$.

□

Proof. The statement (5.65) can be equivalently formulated as follows:

$$\forall i: \quad (\vdash_{\mathcal{G}} \mathbb{C} \wedge \mathbb{C}^q_i \wedge (A^q_i \preceq A) \Rightarrow \mathit{primary}_{\mathcal{G}}(q) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A))$$

Then, all three statements of this lemma have to be proven for all i. Let i be such that

$$\not\vdash_{\mathcal{G}} \mathbb{C} \wedge \mathbb{C}^q_i \wedge (A^q_i \preceq A)$$

For such i, all three statements are trivially satisfied (false premises imply any conclusion). In the rest of the proof, it will be assumed that i is such that

$$\vdash_{\mathcal{G}} \mathbb{C} \wedge \mathbb{C}^q_i \wedge (A^q_i \preceq A)$$

is satisfied. The above condition is equivalent to the following statement:

$$\exists \vec{\alpha}, \vec{\beta}: \quad \mathbb{C}[\vec{\alpha}] \wedge \mathbb{C}^q_i[\vec{\beta}] \wedge A^q_i[\vec{\beta}] \preceq A[\vec{\alpha}]$$

Let $A' \overset{\text{def}}{=} A[\vec{\alpha}]$. Assume (without loss of generality) that $A^q_i = (A^q_{i1}, \ldots, A^q_{in})$. Then $A' = (A'_1, \ldots, A'_n)$ and

$$A^q_{ij}[\vec{\beta}] \preceq A'_j \qquad \text{for all } j \text{ from } 1 \text{ to } n$$

(from the definition of subtyping and the variance of the product). Since A^q_i is an n-arity product, its primary form is also an n-arity product. Let $\mathit{primary}_{\mathcal{G}}(q) = \langle t^q_1, \ldots, t^q_n \rangle$. Note that $t^q_j \in \mathit{Concrete}_{\mathcal{G}}$ from the above definition, the definition of a run-time type, and the fact that $q \in \mathit{RTyp}_{\mathcal{G}}$. Then, $A^t = (A^t_1, \ldots, A^t_n)$ by definition of $\mathit{abstract}_{\mathcal{G}}(\cdot)$.

Each A^q_{ij}, by definition of a run-time type, has a primary functor which belongs to $\mathit{Concrete}_{\mathcal{G}}$. Thus $A^q_{ij} = \underline{c}(r_1, \ldots, r_{\mathit{Arity}(\underline{c})})$, where $\underline{c}$ is a concrete type constructor. Then $t^q_j = \underline{c}$ and therefore $A^t_j = \underline{c}(\ldots)$. At the same time, $A'_j = \bar{c}(\ldots)$ for some $\bar{c}$ and since $A^q_{ij}[\vec{\beta}] \preceq A'_j$ and from the definition of subtyping $\underline{c} \leq_{\mathcal{G}} \bar{c}$. Since $A'_j = c'(\ldots)$, either $A_j = c'(\ldots)$ or $A_j = \alpha_k$ for some k. If $A_j = \alpha_k$, then (since $\mathbb{C}[\vec{\alpha}]$ is true by definition of $\vec{\alpha}$) the following is true:

$$\mathit{pfunctor}(A'_j) \leq_{\mathcal{G}} \mathrm{glb}_{\mathcal{G}}\{c' \mid (\alpha_k \leq c'(\ldots)) \in \mathbb{C}\}$$

and therefore (since $A^q_{ij}[\vec{\beta}] \preceq A'_j$)

$$t^q_j \leq_{\mathcal{G}} \mathrm{glb}_{\mathcal{G}}\{c' \mid (\alpha_k \leq c'(\ldots)) \in \mathbb{C}\}$$

On the other hand, if $A'_j = \bar{c}(\ldots)$, then $t^q_j = \underline{c} \leq_{\mathcal{G}} \bar{c}$. To sum up, the following statement have been shown to be true:

$$t^q_j \leq_{\mathcal{G}} \begin{cases} \mathrm{glb}_{\mathcal{G}}\{c' \mid (\alpha_k \leq c'(\ldots)) \in \mathbb{C}\} & \text{if } A_j \text{ is a variable} \\ \bar{c} & \text{if } A_j = \bar{c}(\ldots) \end{cases} \tag{5.68}$$

and the above case analysis is exhaustive w.r.t. the form of A_j. The above is equivalent to

$$t^q_j \preceq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A_j)$$

and, due to the fact that j was arbitrary, to

$$\mathit{primary}_{\mathcal{G}}(q) = \langle t^q_1, \ldots, t^q_n \rangle \leq_{\mathcal{G}} \langle \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A_1), \ldots, \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A_n) \rangle = \mathit{primary}_{\mathcal{G}}((\mathbb{C}).A) \tag{5.69}$$

which proves (5.65).

In order to show that (5.66) is true, it is sufficient to put

$$\gamma_{jk} = r^i_{jk}$$

where

$$A^q_{ij}[\vec{\beta}] = \underline{c}(r^i_{j1}, \ldots, r^i_{jArity(\underline{c})})$$

Then the following is satisfied (by definitions of $primary_{\mathcal{G}}(\cdot)$, $abstract_{\mathcal{G}}(\cdot)$, and a run-time type):

$$A^q_{ij}[\vec{\beta}] = A^{qp}_j[\vec{\gamma}]$$
$$\mathbb{C}^q_i[\vec{\beta}] \Rightarrow \mathbb{C}^{qp}[\vec{\gamma}]$$

from which

$$A^q_i[\vec{\beta}] = A^{qp}[\vec{\gamma}] \wedge (\mathbb{C}^q_i[\vec{\beta}] \Rightarrow \mathbb{C}^{qp}[\vec{\gamma}])$$

Since $\vec{\alpha}$ and $\vec{\beta}$ were chosen under the only assumption that

$$\mathbb{C}[\vec{\alpha}] \wedge \mathbb{C}^q_i[\vec{\beta}] \wedge A^q_i[\vec{\beta}] \preceq A[\vec{\alpha}]$$

was satisfied, the following is true:

$$\forall \vec{\alpha}, \vec{\beta}: \quad (\mathbb{C}[\vec{\alpha}] \wedge \mathbb{C}^q_i[\vec{\beta}] \wedge A^q_i[\vec{\beta}] \preceq A[\vec{\alpha}] \Rightarrow \exists \vec{\gamma}: \quad A^q_i[\vec{\beta}] = A^{qp}[\vec{\gamma}] \wedge \mathbb{C}^{qp}[\vec{\gamma}])$$

This implies (5.66).

Consider $primary_{\mathcal{G}}(q)$. Since $primary_{\mathcal{G}}(q) \leq_{\mathcal{G}} primary_{\mathcal{G}}((\mathbb{C}).A)$ and according to the definition of $concrete_{\mathcal{G}}(\cdot)$ there are only two possibilities:

1. $primary_{\mathcal{G}}(q) \in concrete_{\mathcal{G}}((\mathbb{C}).A)$ In this case, the statement (5.67) is immediate.
2. $\exists t \in concrete_{\mathcal{G}}((\mathbb{C}).A): \quad primary_{\mathcal{G}}(q) \leq_{\mathcal{G}} t \wedge (\mathbb{C} \wedge \mathbb{C}^{qp} \vdash_{\mathcal{G}} \mathbb{C}^t \wedge A^{qp} \preceq A^t \preceq A)$ where $(\mathbb{C}^t).A^t = abstract_{\mathcal{G}}(t)$. In this case (by (5.66))

   $$\exists t \in concrete_{\mathcal{G}}((\mathbb{C}).A): \quad \mathbb{C} \wedge \mathbb{C}^q_i \wedge (A^q_i \preceq A) \vdash_{\mathcal{G}} \mathbb{C}^{qp} \wedge A^q_i \preceq A^{qp} \preceq A \wedge \mathbb{C}^t \wedge A^{qp} \preceq A^t \preceq A$$

   which implies

   $$\exists t \in concrete_{\mathcal{G}}((\mathbb{C}).A): \quad \mathbb{C} \wedge \mathbb{C}^q_i \wedge (A^q_i \preceq A) \vdash_{\mathcal{G}} \mathbb{C}^t \wedge A^q_i \preceq A^t \preceq A$$

   that is equivalent to (5.67).

□

The following theorem shows that indeed a dispatch of a type-correct behavior application yields a correctly dispatched function.

Theorem 5.8 (Dispatch correctness). If $q \in RTyp_{\mathcal{G}}$ is a run-time type, b is a behavior that satisfies coverage, unambiguity, and correctness conditions,

$$\textbf{behavior } (\mathbb{C}^d_i)\ A^d_i \rightarrow R^d_i \qquad (i \in [1, \ldots, N_d])$$

are the behavior definitions for b,

$$\textbf{association } (\mathbb{C}^a_i)\ A^a_i \rightarrow R^a_i \ \ldots \qquad (i \in [1, \ldots, N_a])$$

are the behavior associations for b,

$$\forall x \in \{a, d\},\ i \in [1, \ldots, N_x]: \quad \vdash_{\mathcal{G}} \mathbb{C}^x_i$$

and

$$\exists x \in \{a, d\},\ i \in [1, \ldots, N_x],\ j: \quad \vdash_{\mathcal{G}} \mathbb{C}^x_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^x_i)$$

then the following holds:

$$k = \mathit{dispatch}_{\mathcal{G}}(b, q) \neq 0 \tag{5.70}$$

$$\forall x \in \{a, d\},\ i \in [1, \ldots, N_x],\ j\colon\quad \mathbb{C}^q_j \wedge \mathbb{C}^x_i \wedge A^q_j \preceq A^x_i \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge A^q_j \preceq A^a_k \preceq A^x_i \tag{5.71}$$

$$\exists j\colon\quad \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^a_k) \tag{5.72}$$

where $q = \mathrm{glb}_j(\mathbb{C}^q_j).A^q_j$ and $t = \mathit{primary}_{\mathcal{G}}(q)$. □

Proof.

Point (5.70) According to definition 4.21, in order to show $k \neq 0$, it is necessary to show that

$$S^b(t) \neq \emptyset \tag{5.73}$$

$$|S^b_{\min}(t)| = 1 \tag{5.74}$$

Point (5.73) There are two possible cases:

1. $\exists i, j\colon\quad \vdash_{\mathcal{G}} \mathbb{C}^a_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^a_i)$ In this case, the statement (5.65) of Lemma 5.5 can be applied to deduce that

   $$t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i)$$

   which ensures that $S^b(t) \neq \emptyset$.
2. $\exists i, j\colon\quad \vdash_{\mathcal{G}} \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i)$. The statement (5.67) of Lemma 5.5 can be applied to deduce that

   $$\exists t \in \mathit{concrete}_{\mathcal{G}}((\mathbb{C}^d_i).A^d_i)\colon\quad \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^t \wedge A^q_j \preceq A^t \preceq A^d_i \tag{5.75}$$

   where $(\mathbb{C}^t).A^t = \mathit{abstract}_{\mathcal{G}}(t)$. From the behavior coverage condition 4.18

   $$\mathbb{C}^t \wedge \mathbb{C}^d_i \wedge (A^t \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^a_l \wedge (A^t \preceq A^a_l \preceq A^d_i)$$

   From this and (5.75)

   $$\mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} A^t \preceq A^a_l \preceq A^d_i \wedge \mathbb{C}^a_l \wedge A^q_j \preceq A^t$$

   Thus

   $$\exists l\colon\quad \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^q_j \wedge A^q_j \preceq A^a_l \preceq A^d_i \wedge \mathbb{C}^a_l \tag{5.76}$$

   and since

   $$\vdash_{\mathcal{G}} \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i)$$

   by the condition of this case the following is true:

   $$\exists l\colon\quad \vdash_{\mathcal{G}} \mathbb{C}^q_j \wedge A^q_j \preceq A^a_l \wedge \mathbb{C}^a_l$$

   The statement (5.65) of Lemma 5.5 is then used to deduce

   $$\exists l\colon\quad t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_l).A^a_l)$$

   which proves the statement under consideration.

Point (5.74) In order to show $|S^b_{\min}(t)| = 1$, it is sufficient to show that $S^b(t) \neq \emptyset$ (which has already been proven) and

$$\begin{aligned} \forall i, j \neq i\colon\quad & t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i) \wedge t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_j).A^a_j) \\ & \Rightarrow \exists k\colon\quad t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_k).A^a_k) \\ & \qquad \wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_k).A^a_k) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i) \\ & \qquad \wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_k).A^a_k) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}^a_j).A^a_j) \end{aligned} \tag{5.77}$$

where $t = \mathit{primary}_{\mathcal{G}}(q)$. From the definition of run-time type

$$q \in \mathit{RTyp}_{\mathcal{G}} \wedge t = \mathit{primary}_{\mathcal{G}}(q) \Rightarrow t \in \mathit{Concrete}_{\mathcal{G}}{}^{n} \text{ for some } n$$

Let t be such that the premises of (5.77) are satisfied. Then,

$$t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \wedge t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \wedge t \in \mathit{Concrete}_{\mathcal{G}}{}^{n}$$

From this and the definition of $\mathit{concrete}_{\mathcal{G}}(\cdot)$

$$\exists u \in \mathit{concrete}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a, (\mathbb{C}_j^a).A_j^a): \quad t \leq_{\mathcal{G}} u$$

From this and the behavior unambiguity condition

$$\begin{aligned}
&\mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \vee \mathit{primary}_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)\\
&\qquad \vee \exists k: \quad u \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)\\
&\qquad\quad \wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)\\
&\qquad\quad \wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)
\end{aligned}$$

There are three possible cases:

1. $\mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a)$. Let $k' = i$. Then, k' satisfies (5.77).
2. $\mathit{primary}_{\mathcal{G}}((\mathbb{C}_j^a).A_j^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)$. Let $k' = j$. Then, k' satisfies (5.77).
3. $\exists k: \quad u \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)$ $\wedge \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)$. In this case, let $k' = k$. Then, k' satisfies (5.77).

In all cases k' that satisfies (5.77) has been found. Thus, $|S^b_{\min}(t)| = 1$ has been shown.

It has been shown that under the conditions of the theorem $|S^b_{\min}(t)| = 1$. Thus the point (5.70) has been proven.

Point (5.71) Assume that x, i, j are such that

$$\nvdash_{\mathcal{G}} \mathbb{C}_j^q \wedge \mathbb{C}_i^x \wedge q \preceq A_i^x$$

Then the statement (5.71) is trivially satisfied. In the rest of the proof it will therefore be assumed that x, i, j are such that

$$\vdash_{\mathcal{G}} \mathbb{C}_j^q \wedge \mathbb{C}_i^x \wedge q \preceq A_i^x$$

is satisfied. There are two possible cases:

1. $x = a$. In this case,

$$\vdash_{\mathcal{G}} \mathbb{C}_j^q \wedge \mathbb{C}_i^a \wedge A_j^q \preceq A_i^a$$

Then $k \neq 0$ (from statement (5.70)). If $k = i$, the statement (5.71) is immediate. Let $k \neq i$. Then

$$t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a) \tag{5.78}$$

from the definition of $\mathit{dispatch}(\cdot,\cdot)$ and $S^b_{\min}(t)$. From the statement (5.66) of Lemma 5.5

$$\mathbb{C}_j^q \wedge \mathbb{C}_i^a \wedge A_j^q \preceq A_i^a \vdash_{\mathcal{G}} \mathbb{C}^{qp} \wedge A_j^q \preceq A^{qp} \preceq A_i^a \tag{5.79}$$

where $\mathit{abstract}_{\mathcal{G}}(\mathit{primary}_{\mathcal{G}}(q)) = (\mathbb{C}^{qp}).A^{qp}$. Let $X_k^u = \mathit{abstract}_{\mathcal{G}}(\mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a))$. Then

$$\begin{gathered}
t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_k^a).A_k^a) \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}(X_k^u)\\
t \leq_{\mathcal{G}} \mathit{primary}_{\mathcal{G}}((\mathbb{C}_i^a).A_i^a)
\end{gathered}$$

(from (5.78) and the definition of $\mathit{primary}_{\mathcal{G}}(\cdot)$ and $\mathit{abstract}_{\mathcal{G}}(\cdot)$) and therefore there are (by definition of $\mathit{concrete}_{\mathcal{G}}(\cdot)$) two possible cases:

(a) $primary_{\mathcal{G}}((\mathbb{C}^{qp}).A^{qp}) \in concrete_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i, X^u_k)$. From this and (5.79)

$$\exists t' \in concrete_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i, X^u_k): \quad \mathbb{C}^q_j \wedge \mathbb{C}^a_i \wedge A^q_j \preceq A^a_i \vdash_{\mathcal{G}} \mathbb{C}^{t'} \wedge A^q_j \preceq A^{t'} \preceq A^a_i \quad (5.80)$$

(choose $t' = primary_{\mathcal{G}}((\mathbb{C}^{qp}).A^{qp})$).

(b) $\exists t'' \in concrete_{\mathcal{G}}((\mathbb{C}^a_i).A^a_i, X^u_k)$: $t \leq_{\mathcal{G}} t'' \wedge (\mathbb{C}^{qp} \wedge \mathbb{C}^a_i \vdash_{\mathcal{G}} \mathbb{C}^{t''} \wedge A^{qp} \preceq A^{t''} \preceq A^a_i)$ where $t = primary_{\mathcal{G}}((\mathbb{C}^{qp}).A^{qp})$ and $(\mathbb{C}^{t''}).A^{t''} = abstract_{\mathcal{G}}(t'')$. From (5.79) and the entailment above

$$\mathbb{C}^q_j \wedge \mathbb{C}^a_i \wedge A^q_j \preceq A^a_i \vdash_{\mathcal{G}} \mathbb{C}^{t''} \wedge A^q_j \preceq A^{qp} \preceq A^{t''} \preceq A^a_i$$

which is equivalent to (5.80) (choose $t' = t''$).

Thus it has been shown that in both cases (5.80) is satisfied. From the behavior correctness condition and (5.78)

$$\mathbb{C}^{t'} \wedge \mathbb{C}^a_i \wedge (A^{t'} \preceq A^a_i) \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge (A^{t'} \preceq A^a_k \preceq A^a_i)$$

From this and (5.80)

$$\mathbb{C}^q_j \wedge \mathbb{C}^a_i \wedge A^q_j \preceq A^a_i \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge A^q_j \preceq A^{t'} \preceq A^a_k \preceq A^a_i$$

which proves statement (5.71) of the theorem for this case.

2. $x = d$. From the proof of the second case of (5.73) (statement (5.76))

$$\exists l: \quad \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^q_j \wedge A^q_j \preceq A^a_l \preceq A^d_i \wedge \mathbb{C}^a_l \quad (5.81)$$

From the previous case (let $x = d, i = l$)

$$\mathbb{C}^q_j \wedge \mathbb{C}^a_l \wedge A^q_j \preceq A^a_l \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge A^q_j \preceq A^a_k \preceq A^a_l$$

From this and (5.81)

$$\exists l: \quad \mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge A^q_j \preceq A^a_k \preceq A^a_l \preceq A^d_i$$

which implies

$$\mathbb{C}^d_i \wedge \mathbb{C}^q_j \wedge (A^q_j \preceq A^d_i) \vdash_{\mathcal{G}} \mathbb{C}^a_k \wedge A^q_j \preceq A^a_k \preceq A^d_i$$

that proves (5.71) in this case.

Point (5.72) Immediate consequence of statement (5.71) and the theorem conditions.

□

In this section it has been proven that in a correctly typechecked expression the dispatch is always valid and correct (no "message not understood" or "message ambiguous" errors). Next section is the proof of the subject reduction theorem.

5.3.5 Subject reduction

The subject reduction theorem that is proven below is the main result of this chapter. It shows that a correctly typechecked program does not produce type or dispatch errors during execution.

Lemma 5.6 (Expanded type propagation). If all types in Θ are expanded (Definition 4.8), E is an rt-environment, $o \in Expressions \cup \mathcal{O}$, and $\Theta \uplus \Theta(\Theta, E) \triangleright o : X$, then X is expanded. □

Proof. By simple induction on the derivation of $\Theta \uplus \Theta(\Theta, E) \rhd o : X$. Follows from the fact that none of the rules in Figure 4.2 and Figure 5.2 can produce non-expanded types provided their premises do not contain them. □

A *consistent triple* defined below can also be termed as a *consistent computation state.*

Definition 5.22 (Consistent triple). A triple $(S, E, \langle \mathbf{expr} \rangle)$ is called *consistent* in a given typing environment Θ if the following conditions are satisfied:

1. $Valid_\Theta(S) = TRUE$
2. $(E(\langle \mathbf{name} \rangle) = \mathbf{a}) \Rightarrow (\Theta \rhd \langle \mathbf{name} \rangle : X_n \;\wedge\; \Theta \rhd \mathbf{a} : X_a \wedge \vdash_{\mathcal{G}} X_a \preceq X_n)$
3. $(\exists o \in Id \cup \mathcal{O}:\;\; \Theta \uplus \Theta(\Theta, E) \rhd o : X) \Rightarrow \;\vdash_{\mathcal{G}} X$
4. $\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{expr} \rangle : X \wedge \vdash_{\mathcal{G}} X$

□

This definition requires that all types of rt-objects that participate in a given environment are valid, and so is the type derived for the expression **expr**. It also requires the validity of the store S.

Finally, the subject reduction theorem can be formulated and proven.

Theorem 5.9 (Subject reduction). If Θ does not contain non-expanded types and a triple $(S, E, \langle \mathbf{expr} \rangle)$ is consistent in Θ then the following holds:

1. The next reduction step for $(S, E, \langle \mathbf{expr} \rangle)$ exists and is unique. The reduction step is understood as a step from the conclusion of the rule to its premise or from one of the premises of the rule to the next one in the order these premises are listed.
2. If in addition $(S, E, \langle \mathbf{expr} \rangle) \Downarrow_{\mathcal{G}} (S', \mathbf{a}')$, then the following holds:
   (a) $Valid_\Theta(S') = TRUE$
   (b) $\Theta \rhd \mathbf{a}' : X'$
   (c) $\vdash_{\mathcal{G}} X' \preceq X$ where $\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{expr} \rangle : X$

□

Proof.

Point 2: The proof is by induction over the derivation of $(S, E, \langle \mathbf{expr} \rangle) \Downarrow_{\mathcal{G}} (S', \mathbf{a}')$. It will be shown that if the theorem is satisfied for the premises of a reduction rule *and* the triple in the conclusion of the rule is consistent, *then* the pair that this triple reduces to according to the conclusion of the rule in question will also satisfy the theorem.

Rule RedAxiom: Since $(S, E, \langle \mathbf{name} \rangle)$ is consistent, $Valid_\Theta(S) = TRUE$. Let $E(\langle \mathbf{name} \rangle) = \mathbf{a}$ ($E(\langle \mathbf{name} \rangle)$ exists since otherwise the rule in question would not be applicable). Then

$$\Theta \rhd \langle \mathbf{name} \rangle : X_n \tag{5.82}$$

$$\Theta \rhd \mathbf{a} : X_a \tag{5.83}$$

$$\vdash_{\mathcal{G}} X_a \preceq X_n \tag{5.84}$$

from the definition of triple consistency. It is left to show that $\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{name} \rangle : X'$ and $\vdash_{\mathcal{G}} X' \preceq X_n$. Since $\Theta(\langle \mathbf{name} \rangle) = X_n$ and $\langle \mathbf{name} \rangle \in Id$, $\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{name} \rangle : X_a$ by definition of $\Theta(\Theta, E)$ and the operation $\uplus$. Thus, $X' = X_a$. From this and equation (5.84)

$$\vdash_{\mathcal{G}} X' \preceq X_n$$

Thus, the pair $(S, \mathbf{a})$ satisfies the conclusions of the theorem.

Rule RedFunction: Immediate from the rule RTClosure in Figure 5.2.

Rule RedProduct: First, it will be shown that if $(S_0, E, (\langle \mathbf{expr} \rangle_1, \ldots, \langle \mathbf{expr} \rangle_m))$ is consistent and $\mathit{Valid}_\Theta(S') = \mathit{TRUE}$, then $(S', E, \langle \mathbf{expr} \rangle_i)$ is consistent. It is sufficient to show that under these conditions

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{expr} \rangle_i : X_i^n \tag{5.85}$$

$$\vdash_{\mathcal{G}} X_i^n \tag{5.86}$$

From the definition of consistency,

$$\Theta \uplus \Theta(\Theta, E) \rhd (\langle \mathbf{expr} \rangle_1, \ldots, \langle \mathbf{expr} \rangle_m) : X^n \tag{5.87}$$

$$\vdash_{\mathcal{G}} X \tag{5.88}$$

Consider the derivation of (5.87). It follows the rules of Figure 4.2, and the last applied rule has to be the rule Product. Thus,

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle \mathbf{expr} \rangle_i : X_i^n \tag{5.89}$$

where

$$X_i^n = \mathrm{glb}_{j_i^n}((\mathbb{C}_i^{n\, j_i^n}).A_i^{n j_i^n}) \tag{5.90}$$

$$X^n = \mathrm{glb}_{j_1^n, \ldots, j_m^n}((\cup_i \mathbb{C}_i^{n\, j_i^n}).(A_1^{n j_1^n}, \ldots, A_m^{n\, j_m^n})) \tag{5.91}$$

Thus, equation (5.85) is proven. Since

$$\begin{aligned}
\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_{j_1^n, \ldots, j_m^n}((\cup_i \mathbb{C}_i^{n\, j_i^n}).(A_1^{n j_1^n}, \ldots, A_m^{n\, j_m^n})) \\
&\equiv \bigwedge_{j_1^n, \ldots, j_m^n} \vdash_{\mathcal{G}} (\cup_i \mathbb{C}_i^{n\, j_i^n}).(A_1^{n j_1^n}, \ldots, A_m^{n\, j_m^n}) \\
&\equiv \bigwedge_{j_1^n, \ldots, j_m^n} \vdash_{\mathcal{G}} \cup_i \mathbb{C}_i^{n\, j_i^n} && (5.92) \\
&\Rightarrow \bigwedge_{j_1^n, \ldots, j_m^n, i} \vdash_{\mathcal{G}} \mathbb{C}_i^{n\, j_i^n} && (5.93) \\
&\equiv \bigwedge_i \bigwedge_{j_i^n} \vdash_{\mathcal{G}} \mathbb{C}_i^{n\, j_i^n} \\
&\equiv \bigwedge_i \bigwedge_{j_i^n} \vdash_{\mathcal{G}} (\mathbb{C}_i^{n\, j_i^n}).A_i^{n j_i^n} && (5.94) \\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} \mathrm{glb}_{j_i^n}(\mathbb{C}_i^{n\, j_i^n}).A_i^{n j_i^n} \\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} X_i^n && (5.95)
\end{aligned}$$

the equation (5.86) is proven as well (equivalences (5.92) and (5.94) are due to the fact that all types are expanded which follows from Lemma 5.6 and the conditions of the theorem). Therefore, $(S', E, \langle \mathbf{expr} \rangle_i)$ is consistent as long as $\mathit{Valid}_\Theta(S') = \mathit{TRUE}$. Since $S_0 = \mathit{TRUE}$ by assumption, $(S_0, E, \langle \mathbf{expr} \rangle_1)$ is consistent. Therefore, by induction assumption, $\mathit{Valid}_\Theta(S_1) = \mathit{TRUE}$. This makes $(S_1, E, \langle \mathbf{expr} \rangle_2)$ consistent and S_2 valid. After $n - 1$ steps, it is shown that all triples $(S_{i-1}, E, \langle \mathbf{expr} \rangle_i)$ are consistent. Then, by induction assumption,

$$\Theta \rhd \mathbf{a}_i : X_i^a \tag{5.96}$$

$$\vdash_{\mathcal{G}} X_i^a \preceq X_i^n \tag{5.97}$$

It is left to show that

$$\Theta \rhd (\mathbf{a}_1, \ldots, \mathbf{a}_n) : X^a \tag{5.98}$$
$$\vdash_{\mathcal{G}} X^a \preceq X^n \tag{5.99}$$
$$\tag{5.100}$$

From equation (5.96) and the rule **RTProduct** (Figure 5.2)

$$\Theta \rhd (\mathbf{a}_1, \ldots, \mathbf{a}_n) : X^a \tag{5.101}$$

where

$$X_i^a = \mathrm{glb}_{j_i^a}((\mathbb{C}_i^{a\,j_i^a}).A_i^{a j_i^a}) \tag{5.102}$$
$$X^a = \mathrm{glb}_{j_1^a,\ldots,j_m^a}((\cup_i \mathbb{C}_i^{a\,j_i^a}).(A_1^{a j_1^a}, \ldots, A_m^{a\,j_m^a})) \tag{5.103}$$

Thus equation (5.98) is proven. Equation (5.99) is equivalent to

$$\begin{aligned}
&\vdash_{\mathcal{G}} \mathrm{glb}_{j_1^a,\ldots,j_m^a}((\cup_i \mathbb{C}_i^{a\,j_i^a}).(A_1^{a j_1^a}, \ldots, A_m^{a\,j_m^a})) \preceq \mathrm{glb}_{j_1^n,\ldots,j_m^n}((\cup_i \mathbb{C}_i^{n\,j_i^n}).(A_1^{n j_1^n}, \ldots, A_m^{n\,j_m^n}))\\
&\equiv \bigvee_{j_1^a,\ldots,j_m^a} \bigwedge_{j_1^n,\ldots,j_m^n} \vdash_{\mathcal{G}} (\cup_i \mathbb{C}_i^{a\,j_i^a}).(A_1^{a j_1^a}, \ldots, A_m^{a\,j_m^a}) \preceq (\cup_i \mathbb{C}_i^{n\,j_i^n}).(A_1^{n j_1^n}, \ldots, A_m^{n\,j_m^n})\\
&\equiv \bigvee_{j_1^a,\ldots,j_m^a} \bigwedge_{j_1^n,\ldots,j_m^n} (\cup_i \mathbb{C}_i^{n\,j_i^n}) \vdash_{\mathcal{G}} \cup_i \mathbb{C}_i^{a\,j_i^a} \cup \{(A_1^{a j_1^a}, \ldots, A_m^{a\,j_m^a}) \preceq (A_1^{n j_1^n}, \ldots, A_m^{n\,j_m^n})\}\\
&\equiv \bigvee_{j_1^a,\ldots,j_m^a} \bigwedge_{j_1^n,\ldots,j_m^n} (\cup_i \mathbb{C}_i^{n\,j_i^n}) \vdash_{\mathcal{G}} (\cup_i \mathbb{C}_i^{a\,j_i^a}) \cup (\cup_i \{A_i^{a j_i^a} \preceq A_i^{n j_i^n}\})\\
&\Leftarrow \bigvee_{j_1^a,\ldots,j_m^a} \bigwedge_{j_1^n,\ldots,j_m^n} (\bigwedge_i \mathbb{C}_i^{n\,j_i^n} \vdash_{\mathcal{G}} \mathbb{C}_i^{a\,j_i^a} \cup \{A_i^{a j_i^a} \preceq A_i^{n j_i^n}\}) &(5.104)\\
&\equiv \bigwedge_i \bigvee_{j_i^a} \bigwedge_{j_i^n} \mathbb{C}_i^{n\,j_i^n} \vdash_{\mathcal{G}} \mathbb{C}_i^{a\,j_i^a} \cup \{A_i^{a j_i^a} \preceq A_i^{n j_i^n}\}\\
&\equiv \bigwedge_i \bigvee_{j_i^a} \bigwedge_{j_i^n} \vdash_{\mathcal{G}} (\mathbb{C}_i^{a\,j_i^a}).A_i^{a j_i^a} \preceq (\mathbb{C}_i^{n\,j_i^n}).A_i^{n j_i^n}\\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} \mathrm{glb}_{j_i^a}((\mathbb{C}_i^{a\,j_i^a}).A_i^{a j_i^a}) \preceq \mathrm{glb}_{j_i^n}((\mathbb{C}_i^{n\,j_i^n}).A_i^{n j_i^n})\\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} X_i^a \preceq X_i^n &(5.105)
\end{aligned}$$

However, the equation (5.105) is the same as (5.97) and therefore the equation (5.99) is proven. The above derivation used equations (5.90) and (5.102). The step (5.104) is a direct consequence of Lemma 5.3 and the fact that the derivation of X_i^n (X_i^a) occurs in a branch different from the one used to derive X_j^n (X_j^a) if $i \neq j$.

Thus it has been shown that $(S_n, (\mathbf{a}_1, \ldots, \mathbf{a}_n))$ satisfies the conclusion of the theorem.

Rule RedLet: First it will be shown that if

$$(S_0, E, \mathtt{let}\ [T]\ \langle\mathbf{name}\rangle\text{=}\langle\mathbf{expr}\rangle_1\ \mathtt{in}\ \langle\mathbf{expr}\rangle_2)$$

is consistent, then so is

$$(S_0, E, \langle\mathbf{expr}\rangle_1)$$

Store validity is immediate; thus it remains to show that

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_1 : X_1^n \tag{5.106}$$
$$\vdash_{\mathcal{G}} X_1^n \tag{5.107}$$

under the conditions that

$$\Theta \uplus \Theta(\Theta, E) \triangleright \texttt{let}\ [T]\ \langle\textbf{name}\rangle{=}\langle\textbf{expr}\rangle_1\ \texttt{in}\ \langle\textbf{expr}\rangle_2 : X^n \tag{5.108}$$

$$\vdash_{\mathcal{G}} X^n \tag{5.109}$$

The outermost rule in the derivation of (5.108) is either **Let** or **TypedLet**. Therefore

$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle\textbf{expr}\rangle_1 : X_1^n = \mathrm{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \tag{5.110}$$

$$\Theta \uplus \Theta(\Theta, E) \uplus \{\langle\textbf{name}\rangle : Y\} \triangleright \langle\textbf{expr}\rangle_2 : X_2^n = \mathrm{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n}) \tag{5.111}$$

where

$$Y = \begin{cases} \mathrm{glb}_i(\mathbb{C}_1^{i\,n}).A_1^{i\,n} & \text{if the rule Let was used} \\ \mathit{expand}_{\mathcal{G}}(T) & \text{if the rule TypedLet was used} \end{cases}$$

which proves (5.106). On the other hand,

$$X^n = \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i).A_2^{j\,n} \tag{5.112}$$

where

$$c_i = \begin{cases} \mathit{TRUE} & \text{if the rule Let was used} \\ (A_1^{i\,n} \preceq \mathit{expand}_{\mathcal{G}}(T)) & \text{if the rule TypedLet was used} \end{cases}$$

and therefore (from (5.109))

$$\begin{aligned} \vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i).A_2^{j\,n} \\ &\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i).A_2^{j\,n} \\ &\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i) \\ &\Rightarrow \bigwedge_i \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \\ &\equiv \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n}).A_1^{i\,n} \\ &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \\ &\equiv \vdash_{\mathcal{G}} X_1^n \end{aligned}$$

which proves (5.107). Therefore,

$$(S_0, E, \langle\textbf{expr}\rangle_1)$$

is consistent and by the induction assumption

$$\mathit{Valid}_{\Theta}(S_1) = \mathit{TRUE} \tag{5.113}$$

$$\Theta \triangleright \mathbf{a}_1 : X_1^a \tag{5.114}$$

$$\vdash_{\mathcal{G}} X_1^a \preceq X_1^n \tag{5.115}$$

From this and the conditions of the theorem it will be shown that

$$(S_1, E \uplus \{\langle\textbf{name}\rangle = \mathbf{a}_1\}, \langle\textbf{expr}\rangle_2)$$

is also consistent. Indeed, S_1 is valid by (5.113). It remains to show that

$$\Theta \uplus \Theta(\Theta, E \uplus \{\langle\textbf{name}\rangle = \mathbf{a}_1\}) \triangleright \langle\textbf{expr}\rangle_2 : X_{a\,2}^n \tag{5.116}$$

$$\vdash_{\mathcal{G}} X_{a\,2}^n \tag{5.117}$$

Consider two possibilities:

Rule Let was applied: In this case, $Y = X_1^n$ and therefore $\vdash_{\mathcal{G}} X_1^n \preceq Y$.
Rule TypedLet was applied: In this case, $Y = expand_{\mathcal{G}}(T)$ and

$$\begin{aligned}\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \text{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq expand_{\mathcal{G}}(T))).A_2^{j\,n} \\ &\Rightarrow \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge (A_1^{i\,n} \preceq expand_{\mathcal{G}}(T)) \\ &\equiv \vdash_{\mathcal{G}} X_1^n \preceq expand_{\mathcal{G}}(T) \\ &\equiv \vdash_{\mathcal{G}} X_1^n \preceq Y\end{aligned}$$

Thus, in both cases $\vdash_{\mathcal{G}} X_1^n \preceq Y$ is satisfied. However, from (5.115), $\vdash_{\mathcal{G}} X_1^a \preceq X_1^n$, and therefore $\vdash_{\mathcal{G}} X_1^a \preceq Y$. This together with (5.114) proves that

$$\Theta \uplus \Theta(\Theta, E \uplus \{\langle\textbf{name}\rangle = \mathbf{a}_1\}) \preceq \Theta \uplus \Theta(\Theta, E) \uplus \{\langle\textbf{name}\rangle : Y\} \tag{5.118}$$

From this, (5.111), and the static subsumption theorem 5.7 the following is true:

1. The statement (5.116) is true.
2. The following statement is true:

$$\vdash_{\mathcal{G}} X_{a\,2}^n \preceq X_2^n \tag{5.119}$$

   from which (5.117) is also true.

Thus,

$$(S_1, E \uplus \{\langle\textbf{name}\rangle = \mathbf{a}_1\}, \langle\textbf{expr}\rangle_2)$$

is consistent and therefore by induction assumption

$$Valid_{\Theta}(S_2) = TRUE \tag{5.120}$$
$$\Theta \triangleright \mathbf{a}_2 : X_2^a \tag{5.121}$$
$$\vdash_{\mathcal{G}} X_2^a \preceq X_{a\,2}^n \tag{5.122}$$

It remains to show that

$$\vdash_{\mathcal{G}} X_2^a \preceq X^n \tag{5.123}$$

where X^n is defined by (5.112). Since $\vdash_{\mathcal{G}} X_2^a \preceq X_{a\,2}^n$ by (5.122) and $\vdash_{\mathcal{G}} X_{a\,2}^n \preceq X_2^n$ by (5.119), it remain to show that $\vdash_{\mathcal{G}} X_2^n \preceq X^n$. Indeed,

$$X^n = \text{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i).A_2^{j\,n}$$
$$X_2^n = \text{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n})$$

and since

$$\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge c_i \vdash_{\mathcal{G}} \mathbb{C}_2^{j\,n} \wedge (A_2^{j\,n} \preceq A_2^{j\,n})$$

for all i, j, by Definition 4.2 $\vdash_{\mathcal{G}} X_2^n \preceq X^n$ which proves (5.123).

Rule RedSequence: First it will be shown that if

$$(S_0, E, \langle\textbf{expr}\rangle_1 \; ; \; \langle\textbf{expr}\rangle_2)$$

is consistent, then so is

$$(S_0, E, \langle\textbf{expr}\rangle_1)$$

Store validity is immediate; thus it remains to show that

$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle \mathbf{expr} \rangle_1 : X_1^n \tag{5.124}$$
$$\vdash_{\mathcal{G}} X_1^n \tag{5.125}$$

under the conditions that

$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle \mathbf{expr} \rangle_1 \; ; \; \langle \mathbf{expr} \rangle_2 : X^n \tag{5.126}$$
$$\vdash_{\mathcal{G}} X^n \tag{5.127}$$

The outermost rule in the derivation of (5.126) is **Seq**; therefore,

$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle \mathbf{expr} \rangle_1 : X_1^n = \mathsf{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \tag{5.128}$$
$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle \mathbf{expr} \rangle_2 : X_2^n = \mathsf{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n}) \tag{5.129}$$

which proves (5.124). On the other hand,

$$X^n = \mathsf{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \tag{5.130}$$

and therefore (from (5.127))

$$\begin{aligned}
\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathsf{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}) \\
&\Rightarrow \bigwedge_i \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n}).A_1^{i\,n} \\
&\equiv \vdash_{\mathcal{G}} \mathsf{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \\
&\equiv \vdash_{\mathcal{G}} X_1^n
\end{aligned}$$

which proves (5.125). Therefore,

$$(S_0, E, \langle \mathbf{expr} \rangle_1)$$

is consistent and by the induction assumption

$$Valid_{\Theta}(S_1) = TRUE \tag{5.131}$$
$$\Theta \triangleright \mathbf{a}_1 : X_1^a \tag{5.132}$$
$$\vdash_{\mathcal{G}} X_1^a \preceq X_1^n \tag{5.133}$$

From this and the conditions of the theorem it will be shown that

$$(S_1, E, \langle \mathbf{expr} \rangle_2)$$

is also consistent. Indeed, S_1 is valid by (5.131). It remains to show that

$$\Theta \uplus \Theta(\Theta, E) \triangleright \langle \mathbf{expr} \rangle_2 : X_2^n \tag{5.134}$$
$$\vdash_{\mathcal{G}} X_2^n \tag{5.135}$$

The statement (5.134) is equivalent to (5.129) and is therefore true. From (5.127)

$$\begin{aligned}\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \\ &\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \\ &\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}) \\ &\Rightarrow \bigwedge_i \vdash_{\mathcal{G}} \mathbb{C}_2^{i\,n} \\ &\equiv \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_2^{i\,n}).A_1^{2n} \\ &\equiv \vdash_{\mathcal{G}} X_2^n\end{aligned}$$

which proves (5.125). Thus

$$(S_1, E, \langle\mathbf{expr}\rangle_2)$$

is consistent and therefore by induction assumption

$$Valid_{\Theta}(S_2) = TRUE \qquad (5.136)$$

$$\Theta \rhd \mathbf{a}_2 : X_2^a \qquad (5.137)$$

$$\vdash_{\mathcal{G}} X_2^a \preceq X_2^n \qquad (5.138)$$

It remains to show that

$$\vdash_{\mathcal{G}} X_2^a \preceq X^n \qquad (5.139)$$

where X^n is defined by (5.130). Since $\vdash_{\mathcal{G}} X_2^a \preceq X_2^n$ by (5.138), it remain to show that $\vdash_{\mathcal{G}} X_2^n \preceq X^n$. Indeed,

$$\begin{aligned}X^n &= \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n}).A_2^{j\,n} \\ X_2^n &= \mathrm{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n})\end{aligned}$$

and since

$$\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \vdash_{\mathcal{G}} \mathbb{C}_2^{j\,n} \wedge (A_2^{j\,n} \preceq A_2^{j\,n})$$

for all i, j, by Definition 4.2 $\vdash_{\mathcal{G}} X_2^n \preceq X^n$ which proves (5.139).

Rule RedFuncApplication: First it will be shown that if

$$(S_0, E, \langle\mathbf{expr}\rangle_1\ \langle\mathbf{expr}\rangle_2)$$

is consistent, then so is

$$(S_0, E, \langle\mathbf{expr}\rangle_1)$$

Store validity is immediate; thus it remains to show that

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_1 : X_1^n \qquad (5.140)$$

$$\vdash_{\mathcal{G}} X_1^n \qquad (5.141)$$

under the conditions that

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_1\ \langle\mathbf{expr}\rangle_2 : X^n \qquad (5.142)$$

$$\vdash_{\mathcal{G}} X^n \qquad (5.143)$$

The outermost rule in the derivation of (5.142) is Appl; therefore,

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_1 : X_1^n = \mathrm{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \tag{5.144}$$

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_2 : X_2^n = \mathrm{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n}) \tag{5.145}$$

which proves (5.140). On the other hand,

$$X^n = \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)).\alpha \tag{5.146}$$

and therefore (from (5.143))

$$\begin{aligned}
\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)).\alpha \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)).\alpha \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)) \\
&\Rightarrow \bigwedge_i \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n}).A_1^{i\,n} \\
&\equiv \vdash_{\mathcal{G}} \mathrm{glb}_i((\mathbb{C}_1^{i\,n}).A_1^{i\,n}) \\
&\equiv \vdash_{\mathcal{G}} X_1^n
\end{aligned}$$

which proves (5.141). Therefore,

$$(S_0, E, \langle\mathbf{expr}\rangle_1) \text{ is consistent} \tag{5.147}$$

and by the induction assumption

$$\mathit{Valid}_\Theta(S_1) = \mathit{TRUE} \tag{5.148}$$

$$\Theta \rhd \mathit{closure}(E', \mathbf{x}, \langle\mathbf{expr}\rangle) : X_1^a \tag{5.149}$$

$$\vdash_{\mathcal{G}} X_1^a \preceq X_1^n \tag{5.150}$$

From this and the conditions of the theorem it will be shown that

$$(S_1, E, \langle\mathbf{expr}\rangle_2)$$

is also consistent. Indeed, S_1 is valid by (5.148). It remains to show that

$$\Theta \uplus \Theta(\Theta, E) \rhd \langle\mathbf{expr}\rangle_2 : X_2^n \tag{5.151}$$

$$\vdash_{\mathcal{G}} X_2^n \tag{5.152}$$

The statement (5.151) is equivalent to (5.145) and is therefore true. From (5.143)

$$\begin{aligned}
\vdash_{\mathcal{G}} X^n &\equiv \vdash_{\mathcal{G}} \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)).\alpha \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)).\alpha \\
&\equiv \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)) \\
&\Rightarrow \bigwedge_j \vdash_{\mathcal{G}} \mathbb{C}_2^{j\,n} \\
&\equiv \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_2^{j\,n}).A_2^{j\,n} \\
&\equiv \vdash_{\mathcal{G}} \mathrm{glb}_j((\mathbb{C}_2^{j\,n}).A_2^{j\,n}) \\
&\equiv \vdash_{\mathcal{G}} X_2^n
\end{aligned}$$

which proves (5.141). Thus

$$(S_1, E, \langle\mathbf{expr}\rangle_2)$$

is consistent and therefore by induction assumption

$$\mathit{Valid}_{\Theta}(S_2) = \mathit{TRUE} \tag{5.153}$$
$$\Theta \rhd \mathbf{a}_2 : X_2^a \tag{5.154}$$
$$\vdash_{\mathcal{G}} X_2^a \preceq X_2^n \tag{5.155}$$

Now it is possible to demonstrate that

$$(S_2, E \uplus E' \uplus \{\mathbf{x} = \mathbf{a}_2\}, \langle\mathbf{expr}\rangle)$$

is consistent. S_2 is valid by (5.153). It has to be shown that

$$\Theta \uplus \Theta(\Theta, E \uplus E' \uplus \{\mathbf{x} = \mathbf{a}_2\}) \rhd \langle\mathbf{expr}\rangle : X_3^n \tag{5.156}$$
$$\vdash_{\mathcal{G}} X_3^n \tag{5.157}$$

From (5.149)

$$\Theta \rhd \mathit{closure}(E', \mathbf{x}, \langle\mathbf{expr}\rangle) : X_1^a$$

and therefore by the rule RTClosure for rt-objects (Figure 5.2)

$$\Theta \uplus \Theta(\Theta, E') \rhd \mathtt{fun}\ \ (\mathtt{x})\ \ \langle\mathbf{expr}\rangle : X_1^a$$

The outermost rule in the above derivation must be the rule Abs and therefore

$$\Theta \uplus \Theta(\Theta, E') \uplus \{\mathbf{x} : \beta\} \rhd \langle\mathbf{expr}\rangle : X_e = \mathrm{glb}_i(\mathbb{C}_e^i).A_e^i \tag{5.158}$$
$$X_1^a = \mathrm{glb}_i((\mathbb{C}_e^i).\beta \to A_e^i) \tag{5.159}$$

where β is a fresh type variable. From (5.143) and (5.146)

$$\vdash_{\mathcal{G}} X^n = \mathrm{glb}_{i,j}(\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \to \alpha)).\alpha$$

and therefore

$$\exists I, J: \quad \vdash_{\mathcal{G}} \mathbb{C}_1^{I\,n} \wedge \mathbb{C}_2^{J\,n} \wedge (A_1^{I\,n} \preceq A_2^{J\,n} \to \alpha) \tag{5.160}$$

By definition of subtyping

$$(X_2^a \preceq \beta) \vdash_{\mathcal{G}} X_2^a \preceq \beta$$

and therefore (since (5.154) is true)

$$(X_2^a \preceq \beta) \vdash_{\mathcal{G}} \Theta \uplus \Theta(\Theta, E \uplus E' \uplus \{\mathbf{x} = \mathbf{a}_2\}) \preceq \Theta \uplus \Theta(\Theta, E') \uplus \{\mathbf{x} : ().\beta\}$$

From this, (5.158) and the static subsumption theorem

$$\Theta \uplus \Theta(\Theta, E \uplus E' \uplus \{\mathbf{x} = \mathbf{a}_2\}) \rhd \langle\mathbf{expr}\rangle : X_3^n \tag{5.161}$$
$$(X_2^a \preceq \beta) \vdash_{\mathcal{G}} X_3^n \preceq X_e \tag{5.162}$$

which implies (5.156). (5.157) is equivalent to

$$\exists k: \quad \vdash_{\mathcal{G}} \mathbb{C}_3^{k\,n}$$

Since (5.162) is equivalent to

$$\forall i, j\, \exists k: \quad \mathbb{C}_2^{i\,a} \wedge (A_2^{i\,a} \preceq \beta) \wedge \mathbb{C}_e^j \vdash_{\mathcal{G}} \mathbb{C}_3^{k\,n} \wedge (A_3^{k\,n} \preceq A_e^j)$$

by definition of subtyping, in order to prove (5.157) it is sufficient to show that

$$\exists i,j:\quad \vdash_{\mathcal{G}} \mathbb{C}_2^{i\,a} \wedge (A_2^{i\,a} \preceq \beta) \wedge \mathbb{C}_e^{j} \tag{5.163}$$

From (5.155)

$$\forall i:\quad \mathbb{C}_2^{i\,n} \vdash_{\mathcal{G}} \bigwedge_j (\mathbb{C}_2^{j\,a} \wedge A_2^{j\,a} \preceq A_2^{j\,n})$$

and from (5.150) and (5.159)

$$\forall i:\quad \mathbb{C}_1^{i\,n} \vdash_{\mathcal{G}} \bigwedge_j (\mathbb{C}_e^{j} \wedge \beta \rightarrow A_e^{i} \preceq A_1^{j\,n})$$

From the above two equations and (5.160)

$$\vdash_{\mathcal{G}} \mathbb{C}_1^{I\,n} \wedge \mathbb{C}_2^{J\,n} \wedge (A_1^{I\,n} \preceq A_2^{J\,n} \rightarrow \alpha) \wedge \bigwedge_j (\mathbb{C}_2^{j\,a} \wedge A_2^{j\,a} \preceq A_2^{j\,n}) \wedge \bigwedge_j (\mathbb{C}_e^{j} \wedge \beta \rightarrow A_e^{I} \preceq A_1^{j\,n})$$

which implies

$$\vdash_{\mathcal{G}} \mathbb{C}_1^{I\,n} \wedge \mathbb{C}_2^{J\,n} \wedge (A_1^{I\,n} \preceq A_2^{J\,n} \rightarrow \alpha) \wedge \mathbb{C}_2^{J\,a} \wedge (A_2^{J\,a} \preceq A_2^{J\,n}) \wedge \mathbb{C}_e^{I} \wedge (\beta \rightarrow A_e^{I} \preceq A_1^{I\,n})$$

which is equivalent (due to the variance of $\rightarrow$ and transitivity of subtyping) to

$$\begin{aligned}\vdash_{\mathcal{G}} \mathbb{C}_1^{I\,n} \wedge \mathbb{C}_2^{J\,n} \wedge \mathbb{C}_2^{J\,a} \wedge \mathbb{C}_e^{I} \wedge (A_1^{I\,n} \preceq A_2^{J\,n} \rightarrow \alpha) \wedge (A_2^{J\,a} \preceq A_2^{J\,n}) \wedge (\beta \rightarrow A_e^{I} \preceq A_1^{I\,n}) \\ \wedge (A_2^{J\,n} \preceq \beta) \wedge (A_e^{I} \preceq \alpha) \wedge (A_2^{J\,a} \preceq \beta)\end{aligned} \tag{5.164}$$

This implies

$$\vdash_{\mathcal{G}} \mathbb{C}_2^{J\,a} \wedge (A_2^{J\,a} \preceq \beta) \wedge \mathbb{C}_e^{I}$$

and therefore (5.163).
Thus

$$(S_2, E \uplus E' \uplus \{\mathbf{x} = \mathbf{a}_2\}, \langle \mathbf{expr} \rangle)$$

is consistent and therefore by induction assumption

$$Valid_{\Theta}(S_3) = TRUE \tag{5.165}$$
$$\Theta \rhd \mathbf{a}_3 : X_3^a \tag{5.166}$$
$$\vdash_{\mathcal{G}} X_3^a \preceq X_3^n \tag{5.167}$$

It remains to show that

$$\vdash_{\mathcal{G}} X_3^a \preceq X^n \tag{5.168}$$

where X^n is defined by (5.146). Since $\vdash_{\mathcal{G}} X_3^a \preceq X_3^n$ by (5.166) it is sufficient to demonstrate that

$$\vdash_{\mathcal{G}} X_3^n \preceq X^n \tag{5.169}$$

From (5.150) by definition of subtyping

$$\forall i \exists i':\quad \mathbb{C}_1^{i\,n} \vdash_{\mathcal{G}} \mathbb{C}_1^{i'\,a} \wedge A_1^{i'\,a} \preceq A_1^{i\,n}$$

which is by (5.159) equivalent to

$$\forall i \exists i':\quad \mathbb{C}_1^{i\,n} \vdash_{\mathcal{G}} \mathbb{C}_e^{i'} \wedge (\beta \rightarrow A_e^{i'} \preceq A_1^{i\,n})$$

From the above and the definition of subtyping, the following is true:

$$\forall i, j\,\exists i'\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}} \mathbb{C}_e^{i'} \wedge (\beta \rightarrow A_e^{i'} \preceq A_1^{i\,n})$$

Using the properties of entailment

$$\forall i, j\,\exists i'\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}}$$
$$\mathbb{C}_e^{i'} \wedge \mathbb{C}_2^{j\,n} \wedge (\beta \rightarrow A_e^{i'} \preceq A_1^{i\,n}) \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha)$$

from which (by transitivity of subtyping)

$$\forall i, j\,\exists i'\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}} \mathbb{C}_e^{i'} \wedge \mathbb{C}_2^{j\,n} \wedge (\beta \rightarrow A_e^{i'} \preceq A_2^{j\,n} \rightarrow \alpha)$$

and by variance of the type constructor $\rightarrow$

$$\forall i, j\,\exists i'\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}} \mathbb{C}_e^{i'} \wedge \mathbb{C}_2^{j\,n} \wedge A_2^{j\,n} \preceq \beta \wedge A_e^{i'} \preceq \alpha \qquad (5.170)$$

On the other hand, from (5.155) and (5.162) it follows that

$$(X_2^n \preceq \beta) \vdash_{\mathcal{G}} X_3^n \preceq X_e$$

which is equivalent to

$$\forall i, j\,\exists k\colon \quad \mathbb{C}_2^{j\,n} \wedge (A_2^{j\,n} \preceq \beta) \wedge \mathbb{C}_e^{i} \vdash_{\mathcal{G}} \mathbb{C}_3^{k\,n} \wedge (A_3^{k\,n} \preceq A_e^{i})$$

from which and (5.170)

$$\forall i, j\,\exists i', k\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}}$$
$$\mathbb{C}_e^{i'} \wedge \mathbb{C}_2^{j\,n} \wedge A_2^{j\,n} \preceq \beta \wedge A_e^{i'} \preceq \alpha \wedge \mathbb{C}_3^{k\,n} \wedge (A_3^{k\,n} \preceq A_e^{i'})$$

which implies (by transitivity)

$$\forall i, j\,\exists i', k\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}}$$
$$\mathbb{C}_e^{i'} \wedge \mathbb{C}_2^{j\,n} \wedge A_2^{j\,n} \preceq \beta \wedge A_e^{i'} \preceq \alpha \wedge \mathbb{C}_3^{k\,n} \wedge (A_3^{k\,n} \preceq A_e^{i'}) \wedge (A_3^{k\,n} \preceq \alpha)$$

which, in turn, implies

$$\forall i, j\,\exists k\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \vdash_{\mathcal{G}} \mathbb{C}_3^{k\,n} \wedge (A_3^{k\,n} \preceq \alpha)$$

which is equivalent to

$$\vdash_{\mathcal{G}} X_3^n \preceq X^n$$

due to (5.146). Thus (5.169) and therefore (5.168) are proven.

Rule RedBehApplication: The first two premises of this rule are the same as those in the rule **RedFuncApplication**. Therefore, the equations (5.141)–(5.155) are proven in the same way. Since $\langle$**expr**$\rangle_1$ reduces to a behavior (**behavior**), the following is true:

$$\Theta \uplus \Theta(\Theta, E) \triangleright \mathbf{behavior} : X_1^a$$

where

$$X_1^a = \mathrm{glb}_i(\mathbb{C}_1^{i\,a}).A_1^{i\,a}$$

and

$$A_1^{i\,a} = A_b^i \rightarrow_b R_b^i$$

From this and (5.150)

$$X_1^a \preceq X_1^n \equiv \forall i\, \exists l\colon\ \ \mathbb{C}_1^{i\,n} \vdash_{\mathcal{G}} \mathbb{C}_1^{l\,a} \wedge (A_1^{l\,a} \preceq A_1^{i\,n})$$
$$\equiv \forall i\, \exists l\colon\ \ \mathbb{C}_1^{i\,n} \vdash_{\mathcal{G}} \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \qquad (5.171)$$

From (5.143) and (5.146)

$$\forall i, j\colon\ \ \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha))$$

and from this and (5.171)

$$\forall i, j\, \exists l\colon\ \ \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge (A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n})$$

from which (due to variance of $\rightarrow$ and $\rightarrow_b$ and their subtyping relationship in $\mathcal{G}$)

$$\forall i, j\, \exists l\colon\ \ \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge (A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha) \qquad (5.172)$$

From (5.155)

$$\forall j\, \exists m\colon\ \ \mathbb{C}_2^{j\,n} \vdash_{\mathcal{G}} \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \qquad (5.173)$$

and from this and (5.172)

$$\forall i, j\, \exists l, m\colon\ \ \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha)$$

which implies

$$\exists l, m\colon\ \ \vdash_{\mathcal{G}} \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_b^l)$$

From this and the dispatch correctness theorem 5.8

$$k = \mathit{dispatch}_{\mathcal{G}}(\texttt{behavior}, \mathtt{a}_2) \neq 0 \qquad (5.174)$$
$$\forall l, m\colon\ \ \mathbb{C}_2^{m\,a} \wedge \mathbb{C}_1^{l\,a} \wedge A_2^{m\,a} \preceq A_b^l \vdash_{\mathcal{G}} \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \qquad (5.175)$$
$$\exists m\colon\ \ \vdash_{\mathcal{G}} \mathbb{C}_1^{k\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_b^k) \qquad (5.176)$$

There are two possibilities: the dispatch might produce a function or a primitive. The first case is handled by the rule **RedBehApplication**, while the second is handled by the rule **RedBehPrimApplication**. Here only the first alternative is considered; the second one will be considered later on. Assume that the dispatch produced a function F. Then, by the functional consistency condition,

$$\Theta_0 \rhd \mathtt{F} : X_f' \qquad (5.177)$$
$$\vdash_{\mathcal{G}} X_f' \preceq (\mathbb{C}_1^{k\,a}).(A_b^k \rightarrow R_b^k) \qquad (5.178)$$

Since $\Theta \preceq \Theta_0$ for any valid Θ, by the static subsumption theorem

$$\Theta \uplus \Theta(E, \Theta) \rhd \mathtt{F} : X_f = \mathrm{glb}_p(\mathbb{C}_f^p).A_f^p \qquad (5.179)$$
$$\vdash_{\mathcal{G}} X_f \preceq X_f' \qquad (5.180)$$

and therefore

$$\vdash_{\mathcal{G}} X_f \preceq (\mathbb{C}_1^{k\,a}).(A_b^k \rightarrow R_b^k) \qquad (5.181)$$

By using the rule **Appl** it is concluded that

$$\Theta \uplus \Theta(E, \Theta) \rhd (\ \mathtt{F}\ \mathtt{a}_2\ ) : X_3^n = \mathrm{glb}_{m,p}(\mathbb{C}_2^{m\,a} \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_2^{m\,a} \rightarrow \gamma).\gamma \qquad (5.182)$$

where γ occurs in X_3^a only in places denoted explicitly above (as it is fresh when the rule **Appl** is applied).
It is now left to show that

$$(S_2, E, \texttt{(fun(x) } \langle\texttt{expr}\rangle \texttt{) a}_2)$$

is consistent and that the result type conforms to the third condition of the theorem. S_2 is valid by (5.153). In order to show triple consistency it is left to demonstrate that

$$\Theta \uplus \Theta(\Theta, E) \rhd \texttt{(F a}_2\texttt{)} : X_3^n \tag{5.183}$$

$$\vdash_{\mathcal{G}} X_3^n \tag{5.184}$$

where $\texttt{F} \overset{\text{def}}{=} \texttt{(fun (x) } \langle\texttt{expr}\rangle \texttt{)}$.
(5.183) follows from (5.182). (5.184) is equivalent to

$$\exists m, p:\quad \vdash_{\mathcal{G}} \mathbb{C}_2^{m\,a} \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_2^{m\,a} \rightarrow \gamma \tag{5.185}$$

From (5.181)

$$\exists p:\quad \mathbb{C}_1^{k\,a} \vdash_{\mathcal{G}} \mathbb{C}_f^p \wedge A_f^p \preceq A_b^k \rightarrow R_b^k$$

and from this and (5.176)

$$\exists m, p:\quad \vdash_{\mathcal{G}} \mathbb{C}_1^{k\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_b^k) \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_b^k \rightarrow R_b^k$$

which implies (5.185) (let $\gamma = R_b^k$). Thus the triple

$$(S_2, E, \texttt{(fun(x) } \langle\texttt{expr}\rangle \texttt{) a}_2)$$

is consistent and therefore (by induction assumption)

$$Valid_{\Theta}(S_3) = TRUE \tag{5.186}$$

$$\Theta \rhd \texttt{a}_3 : X_3^a \tag{5.187}$$

$$\vdash_{\mathcal{G}} X_3^a \preceq X_3^n \tag{5.188}$$

In order to complete the proof of this case, it is necessary to show that

$$\vdash_{\mathcal{G}} X_3^a \preceq X^n$$

Since $\vdash_{\mathcal{G}} X_3^a \preceq X_3^n$ by (5.188), it is sufficient to show that

$$\vdash_{\mathcal{G}} X_3^n \preceq X^n$$

which is equivalent to

$$\forall i, j\, \exists m, p:\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \mathbb{C}_2^{m\,a} \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_2^{m\,a} \rightarrow \gamma \wedge \gamma \preceq \alpha \tag{5.189}$$

By the definition of subtyping,

$$\forall i, j:\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha$$

From this and (5.171)

$$\forall i, j\, \exists l:\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}}$$
$$\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n})$$

from which and (5.173)

$$\begin{aligned}\forall i,j\,\exists l,m:\quad & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n})\end{aligned}$$

from which and (5.175)

$$\begin{aligned}\forall i,j\,\exists l,m:\quad & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \\ & \qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l\end{aligned}$$

From this and (5.181)

$$\begin{aligned}\forall i,j\,\exists l,m,p:\quad & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \\ & \qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_b^k \rightarrow R_b^k\end{aligned}$$

From this and the global behavior consistency condition 4.10

$$\begin{aligned}\forall i,j\,\exists l,m,p:\quad & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ & \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \\ & \qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_b^k \rightarrow R_b^k \wedge R_b^k \preceq R_b^l\end{aligned}$$

From this, transitivity, and relationship between $\rightarrow$ and $\rightarrow_b$

$$\forall i,j\,\exists l,m:\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \mathbb{C}_2^{m\,a} \wedge \mathbb{C}_f^p \wedge A_f^p \preceq A_2^{m\,a} \rightarrow R_b^l \wedge R_b^l \preceq \alpha$$

which implies (5.189) (put $\gamma = R_b^l$).

Rule RedBehPrimApplication: This rule is identical to **RedBehPrimApplication** up to the point where the dispatched function is applied. Therefore, the equations (5.141)–(5.155) and (5.171)–(5.176) are proven in the same way.

It now remains to show that under these conditions

$$\mathit{primitive}_i(S_2, \mathsf{a}_2)$$

is defined, the resulting store S_3 is consistent, and the type of the resulting object is less then the type derived initially for the expression:

$$\Theta \triangleright \mathsf{a}_3 : X_3^a = \mathsf{glb}_m(\mathbb{C}_3^{m\,a}).A_3^{m\,a} \tag{5.190}$$

$$\vdash_{\mathcal{G}} X_3^a \preceq X^n \tag{5.191}$$

S_2 is valid by (5.153). Since

$$\begin{aligned}\Theta \triangleright \mathsf{a}_2 : X_2^a = \mathsf{glb}_m(\mathbb{C}_2^{m\,a}).A_2^{m\,a} \\ \exists m:\quad \vdash_{\mathcal{G}} \mathbb{C}_1^{k\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_b^k)\end{aligned}$$

(by (5.154) and (5.176) respectively), and

$$\mathit{primitive}_i(S_2, \mathsf{a}_2) = (S_3, \mathsf{a}_3) \tag{5.192}$$

$$\mathit{Valid}_\Theta(S_3) = \mathit{TRUE} \tag{5.193}$$

$$\Theta \triangleright \mathsf{a}_3 : X_3^a = \mathsf{glb}_p(\mathbb{C}_3^{j\,a}).A_3^{p\,a} \tag{5.194}$$

$$\forall m\,\exists p:\quad \mathbb{C}_1^{k\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_b^k) \vdash_{\mathcal{G}} \mathbb{C}_3^{p\,a} \wedge (A_3^{p\,a} \preceq R_b^k) \tag{5.195}$$

by the primitive function correctness condition 5.21. Now it is left to show that

$$\vdash_{\mathcal{G}} X_3^a \preceq X^n$$

which is equivalent to

$$\forall i, j\, \exists p\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \mathbb{C}_3^{p\,a} \wedge A_3^{p\,a} \preceq \alpha \tag{5.196}$$

From the definition of subtyping

$$\forall i, j\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha$$

From this and (5.171)

$$\begin{aligned} &\forall i, j\, \exists l\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\qquad\qquad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \end{aligned}$$

From this and (5.173)

$$\begin{aligned} &\forall i, j\, \exists l, m\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \wedge \mathbb{C}_1^{l\,a} \wedge (A_b^l \rightarrow_b R_b^l \preceq A_1^{i\,n}) \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \end{aligned}$$

From the above, the transitivity of subtyping, relationship between $\rightarrow$ and $\rightarrow_b$ and their variance

$$\begin{aligned} &\forall i, j\, \exists l, m\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\qquad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \wedge (A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha) \end{aligned}$$

from which and (5.175)

$$\begin{aligned} &\forall i, j\, \exists l, m\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\qquad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \wedge (A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha) \\ &\qquad\qquad\qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \end{aligned}$$

From this and (5.195)

$$\begin{aligned} &\forall i, j\, \exists l, m, p\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\qquad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \wedge (A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha) \\ &\qquad\qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \wedge \mathbb{C}_3^{p\,a} \wedge (A_3^{p\,a} \preceq R_b^k) \end{aligned}$$

From the above and the global behavior consistency condition 4.10

$$\begin{aligned} &\forall i, j\, \exists l, m, p\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\qquad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \wedge (A_2^{j\,n} \preceq A_b^l) \wedge (R_b^l \preceq \alpha) \\ &\qquad\qquad \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \wedge \mathbb{C}_3^{p\,a} \wedge (A_3^{p\,a} \preceq R_b^k) \wedge R_b^k \preceq R_l^k \end{aligned}$$

from which and the transitivity of subtyping

$$\begin{aligned} &\forall i, j\, \exists l, m, p\colon \quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha \vdash_{\mathcal{G}} \\ &\quad \mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge \mathbb{C}_1^{l\,a} \wedge \mathbb{C}_2^{m\,a} \wedge (A_2^{m\,a} \preceq A_2^{j\,n}) \wedge (A_2^{j\,n} \preceq A_b^l) \wedge \mathbb{C}_1^{k\,a} \wedge A_2^{m\,a} \preceq A_b^k \preceq A_b^l \\ &\qquad\qquad\qquad \wedge \mathbb{C}_3^{p\,a} \wedge A_3^{p\,a} \preceq \alpha \end{aligned}$$

which implies (5.196).

Point 1: Since the triple $(S, E, \langle\mathbf{expr}\rangle)$ is consistent, $\langle\mathbf{expr}\rangle$ is syntactically valid. The reduction rules are structural with the following exceptions:

1. Rule **RedAxiom** additionally requires that $E(\langle\mathbf{name}\rangle) \neq \emptyset$
2. An application expression can be processed by one of the rules **RedFuncApplication**, **RedBehApplication**, or **RedBehPrimApplication**. These rules have additional (non-structural) premises.

Assume that $\langle\mathbf{expr}\rangle = \langle\mathbf{name}\rangle$ (as is required for the rule **RedAxiom** to be applicable). Assume $E(\langle\mathbf{name}\rangle) = \emptyset$. Then $\Theta \uplus \Theta(E, \Theta) \triangleright \langle\mathbf{name}\rangle : \top$ which contradicts the assumption about triple consistency (since $\nvdash_{\mathcal{G}} \top$). Thus, the rule **RedAxiom** works for all correctly type-checked names.

Assume that $\langle\mathbf{expr}\rangle = \langle\mathbf{expr}\rangle_1 \langle\mathbf{expr}\rangle_2$. The first premise of all three rules is identical. It will be shown that under these conditions, the first expression, if it reduces, reduces to either a closure (in which case the rule **RulFunApplication** is applicable and its second premise is checked) or a behavior (in this case an additional analysis is needed). Indeed, by triple consistency

$$\begin{aligned} \vdash_{\mathcal{G}} X^n \equiv & \bigwedge_i \bigwedge_j \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \mathbb{C}_2^{j\,n} \wedge A_1^{i\,n} \preceq A_2^{j\,n} \rightarrow \alpha) \\ \Rightarrow & \bigwedge_i \vdash_{\mathcal{G}} (\mathbb{C}_1^{i\,n} \wedge \wedge A_1^{i\,n} \preceq \beta \rightarrow \alpha) \end{aligned} \tag{5.197}$$

where β is a fresh type variable and equation (5.144) is satisfied. Also from triple consistency (5.147) is true and therefore

$$(S_0, E, \langle\mathbf{expr}\rangle_1) \Downarrow_{\mathcal{G}} (S_1, \mathbf{a}_1) \tag{5.198}$$

$$Valid_{\Theta}(S_1) = TRUE \tag{5.199}$$

$$\Theta \triangleright \mathbf{a}_1 : X_1^a = \mathrm{glb}_l(\mathbb{C}_1^{l\,a}).A_1^{l\,a} \tag{5.200}$$

$$\vdash_{\mathcal{G}} X_1^a \preceq X_1^n \tag{5.201}$$

From (5.197), (5.201), and transitivity

$$\forall i \exists l: \quad \vdash_{\mathcal{G}} A_1^{l\,a} \preceq \beta \rightarrow \alpha$$

Since X_1^a is a concrete type, the primary functor of X_1^a must be concrete. The only concrete types c such that $(c \preceq_{\mathcal{G}} \rightarrow)$ are $(\rightarrow_b)$ and $(\rightarrow)$ itself. Thus, the primary functor of X_1^a is either $(\rightarrow)$ or $(\rightarrow_b)$. The only rt-objects that can have a type with a primary functor of $(\rightarrow)$ are closures. At the same time the only rt-objects that can have a type with a primary functor of $(\rightarrow_b)$ are behaviors. Thus, $\mathbf{a}_1$ must be either a closure (in which case the rule **RedFuncApplication** is applicable) or a behavior.

If $\mathbf{o}$ is a behavior, its dispatch (fourth premise of the rules **RedBehApplication** and **RedBehPrimApplication**) can:

1. Produce a function. In this case, the rule **RedBehApplication** will work.
2. Produce a primitive. In this case, the rule **RedBehPrimApplication** will work.
3. Produce an error (0). In this case, no rule is applicable.

Thus it is sufficient to show that under these conditions, the dispatch will never yield 0. However, this statement has already been proven in (5.174).

Thus, the statement 1 of the theorem has also been proven.

□

In this section, the main result of this chapter (the subject reduction theorem) has been proven. The next section describes incorporation of imperative types into the type system developed so far.

5.4 Imperative types

In this section, it will be shown that it is possible to consistently introduce state in the framework described above. Interestingly, state does not require any special type treatment. The set of appropriate primitive functions along with a particular treatment of store and store validity is sufficient.

This is in sharp contrast to ML-based type system where imperative types (type constructor **ref**) are treated specially by the type system. This special treatment has two main disadvantages:

1. It restricts the use of imperative types by applying less permissive type discipline to them.
2. It significantly complicates the development of a system with several different imperative types, as for such a system the base type theory has to be modified.

The family of imperative types **T_Var(X)** is introduced as follows:

```
type T_Var(novar X);
behavior T_Var(X) -> X get : primitive1;
behavior (T_Var(X),X) -> T_Unit set : primitive2;
```

with the additional constraint that there are no concrete types below **T_Var**.

Let

$$\mathcal{V} = \{\mathsf{o} \mid \mathsf{o} \in \mathcal{O} \wedge \Theta_0 \rhd \mathsf{o} : \mathtt{T\_Var}(\ldots)\}$$

be the set of all objects with imperative types. The store S is then defined as a partial function from $\mathcal{V}$ into $\mathcal{O}$. For each "variable" (an object of type **T_Var**) the store gives its "contents" (an object stored in it).

The store validity predicate $\mathit{Valid}_\Theta(S)$ is defined in the following manner:

$$\begin{aligned}\mathit{Valid}_\Theta(S) \overset{\text{def}}{\Longleftrightarrow} (\Theta \rhd \mathbf{v} : V = \mathrm{glb}_i(\mathbb{C}_i^v).V_i \wedge \exists i{:}\quad & \vdash_{\mathcal{G}} \mathbb{C}_i^v \wedge V_i \preceq \mathtt{T\_Var}(\alpha) \Rightarrow \\ & S(\mathbf{v}) = \mathsf{o} \\ & \wedge\, \Theta \rhd \mathsf{o} : O = \mathrm{glb}_j(\mathbb{C}_j^o).O_j \\ & \wedge\, \forall i\, \exists j{:}\quad \mathbb{C}_i^v \wedge V_i \preceq \mathtt{T\_Var}(\alpha) \vdash_{\mathcal{G}} \mathbb{C}_j^o \wedge O_j \preceq \alpha)\end{aligned} \tag{5.202}$$

An intuitive meaning of the above definition is that a store is valid if for every variable there is a value stored in it and that value is of the same (or smaller) type as the type of the variable.

The semantics of primitive functions is then defined as follows:

$$\mathit{primitive}_1(S, \mathbf{v}) = (S, S(\mathbf{v})) \tag{5.203}$$

(getting the value of a variable **v**) and

$$\mathit{primitive}_2(S, (\mathbf{v}, \mathbf{a})) = (S \uplus (S(\mathbf{v}) = \mathbf{a}), \mathbf{unit}) \tag{5.204}$$

(setting the value of a variable **v** to **a**). In the latter case

$$(S \uplus (S(\mathbf{v}) = \mathbf{a}))(\mathsf{o}) \overset{\text{def}}{=} \begin{cases} S(\mathsf{o}) & \text{if } \mathsf{o} \neq \mathbf{v} \\ \mathbf{a} & \text{if } \mathsf{o} = \mathbf{v} \end{cases}$$

In order to show that this definition is valid and that in the system extended with imperative types in this manner the subject reduction theorem still holds, it is sufficient to demonstrate that the above primitives obey the primitive function correctness condition 5.21.

Getting a value: It is necessary to show that if S is a store and **v** is an rt-object such that

$$\mathit{Valid}_\Theta(S) = \mathit{TRUE} \tag{5.205}$$

$$\Theta \rhd \mathbf{v} : X^v, X^v = \mathrm{glb}_i(\mathbb{C}_i^v).A_i \tag{5.206}$$

$$\exists i{:}\quad \vdash_{\mathcal{G}} \mathbb{C}_i^v \wedge A_i \preceq \mathtt{T\_Var}(\alpha) \tag{5.207}$$

then the following holds:

$$primitive_1(S, \mathbf{v}) = (S, \mathbf{a}) \text{ where } \mathbf{a} = S(\mathbf{v}) \tag{5.208}$$
$$Valid_\Theta(S) = TRUE \tag{5.209}$$
$$\Theta \triangleright \mathbf{a} : X^a = \text{glb}_j(\mathbb{C}_j^a).R_j \tag{5.210}$$
$$\forall i \exists j: \quad \mathbb{C}_i^v \wedge (A_i \preceq \texttt{T\_Var}(\alpha)) \vdash_{\mathcal{G}} \mathbb{C}_j^a \wedge (R_j \preceq \alpha) \tag{5.211}$$

In order to show (5.208), it is sufficient to demonstrate that $S(\mathbf{v})$ exists. From (5.207), the fact that X^v is an rt-type, and the condition that there are no concrete types below **T_Var** it follows that

$$X^v = \texttt{T\_Var}(T) \tag{5.212}$$

where T is some type expression. From this and the validity of S (by (5.202))

$$S(\mathbf{v}) = \mathbf{a} \tag{5.213}$$
$$\Theta \triangleright \mathbf{a} : X^a = \text{glb}_j(\mathbb{C}_j^a).R_j \tag{5.214}$$
$$\exists j: \quad \texttt{T\_Var}(T) \preceq \texttt{T\_Var}(\alpha) \vdash_{\mathcal{G}} \mathbb{C}_j^a \wedge (R_j \preceq \alpha) \tag{5.215}$$

By (5.213) $S(\mathbf{v})$ exists, and therefore (5.208) is proven. (5.210) is equivalent to (5.214). The validity of S is trivial by (5.205) (getting a value does not change the store). (5.211) is equivalent to (5.215) and is therefore proven.

Thus, consistency of the first primitive (getting a value) has been demonstrated.

Setting a value: It is necessary to show that if S is a store and $\mathbf{p}$ is an rt-object such that

$$Valid_\Theta(S) = TRUE \tag{5.216}$$
$$\Theta \triangleright \mathbf{p} : X^p, X^p = \text{glb}_i(\mathbb{C}_i^p).P_i \tag{5.217}$$
$$\exists i: \quad \vdash_{\mathcal{G}} \mathbb{C}_i^p \wedge P_i \preceq (\texttt{T\_Var}(\alpha), \alpha) \tag{5.218}$$

then the following holds:

$$primitive_2(S, \mathbf{p}) = (S', \texttt{unit}) \tag{5.219}$$
$$Valid_\Theta(S') = TRUE \tag{5.220}$$
$$\Theta \triangleright \texttt{unit} : X^u \tag{5.221}$$
$$\forall i: \quad \mathbb{C}_i^p \wedge P_i \preceq (\texttt{T\_Var}(\alpha), \alpha) \vdash_{\mathcal{G}} X^u \preceq \texttt{T\_Unit} \tag{5.222}$$

The statements (5.221) and (5.222) are immediately true as $\Theta_0 \triangleright \texttt{unit} : \texttt{T\_Unit}$. Thus it remains to show the validity of (5.219) and (5.220).

Since X^p is a run-time type and there are no concrete types below **T_Product$_2$**, the following is true:

$$\forall i: \quad P_i = (V_i, S_i) \tag{5.223}$$

At the same time, from (5.223) and the definition of a run-time object

$$\mathbf{p} = (\ \mathbf{v},\ \mathbf{s}\ ) \tag{5.224}$$

On the other hand, from (5.218) and (5.223) it follows that

$$\exists i: \quad \vdash_{\mathcal{G}} \mathbb{C}_i^p \wedge V_i \preceq \texttt{T\_Var}(\alpha) \tag{5.225}$$

and from the rule **RTProduct**

$$\Theta \triangleright \mathbf{v} : X^v = \text{glb}_l(\mathbb{C}_l^v).V_l' \tag{5.226}$$
$$\Theta \triangleright \mathbf{s} : X^s = \text{glb}_m(\mathbb{C}_m^s).S_m' \tag{5.227}$$
$$X^p = \text{glb}_{l,m}(\mathbb{C}_l^v \wedge \mathbb{C}_m^s).(V_l', S_m') \tag{5.228}$$

From this and (5.217)

$$P_i = (V_l', S_m') \text{ for some } l, m$$
$$\mathbb{C}_i^p = \mathbb{C}_l^v \wedge \mathbb{C}_m^s \text{ for some } l, m$$

from which and (5.223)

$$V_i = V_l' \text{ for some } l$$
$$S_i = S_m' \text{ for some } m$$

Let us denote the indices that satisfy the above equations as l_i and m_i. To sum up, the following statements have been proven:

$$\Theta \triangleright \mathbf{p} : X^p = \mathrm{glb}_i(\mathbb{C}_{l_i}^v \wedge \mathbb{C}_{m_i}^s).(V_{l_i}, S_{m_i}) \tag{5.229}$$
$$\mathbf{p} = (\ \mathtt{v},\ \ \mathtt{s}\ ) \tag{5.230}$$
$$\Theta \triangleright \mathtt{v} : X^v = \mathrm{glb}_i(\mathbb{C}_{l_i}^v).V_{l_i} \tag{5.231}$$
$$\Theta \triangleright \mathtt{s} : X^s = \mathrm{glb}_i(\mathbb{C}_{m_i}^s).S_{m_i} \tag{5.232}$$

From this, (5.225), and the fact that there are no concrete types below **T_Var** it follows that

$$\forall i: \quad V_{l_i} = \mathbf{T\_Var}(U_i) \tag{5.233}$$

From this and the semantics of *primitive*$_2$

$$primitive_2(S, \mathbf{p}) = (S', \mathtt{unit}) \tag{5.234}$$
$$\text{where } S' = S \uplus (S(\mathtt{v}) = \mathtt{s}) \tag{5.235}$$

Thus, (5.220) is true. It remains to show that S' is valid. Since S is valid and S' can only be different from it on $\mathtt{v}$, it is sufficient to demonstrate that

$$\forall l\, \exists m: \quad \mathbb{C}_l^v \wedge V_l' \preceq \mathbf{T\_Var}(\alpha) \vdash_{\mathcal{G}} \mathbb{C}_m^s \wedge S_m' \preceq \alpha \tag{5.236}$$

X^v is a run-time type with primary functor **T_Var** (from (5.233)) and therefore $FTV(U_i) = \emptyset$ for all i. Also from the definition of a run-time type $U_i = U$ for some U and all i. From this and satisfiability of run-time type constraints $(\mathbb{C}_l^v).V_l' = \mathbf{T\_Var}(U)$. Thus (5.236) (that we are attempting to prove) is equivalent to

$$\exists m: \quad \mathbf{T\_Var}(U) \preceq \mathbf{T\_Var}(\alpha) \vdash_{\mathcal{G}} \mathbb{C}_m^s \wedge S_m' \preceq \alpha$$

which is equivalent to

$$\exists m: \quad \alpha = U \vdash_{\mathcal{G}} \mathbb{C}_m^s \wedge S_m' \preceq \alpha$$

due to novariance of **T_Var**. This is in turn equivalent to

$$\exists m: \quad \vdash_{\mathcal{G}} \mathbb{C}_m^s \wedge S_m' \preceq U \tag{5.237}$$

by definition of entailment and the fact that $\alpha \notin FTV((\mathbb{C}_m^s).S_m')$ and $FTV(U) = \emptyset$.

At the same time, (5.218) (the condition) is equivalent to

$$\exists i: \quad \vdash_{\mathcal{G}} \mathbb{C}_{m_i}^s \wedge (\mathbf{T\_Var}(U), S_{m_i}) \preceq (\mathbf{T\_Var}(\alpha), \alpha)$$

which implies (by variance of product and definition of m_i)

$$\exists m: \quad \vdash_{\mathcal{G}} \mathbb{C}_m^s \wedge \mathbf{T\_Var}(U) \preceq \mathbf{T\_Var}(\alpha) \wedge S_m \preceq \alpha$$

which is equivalent to

$$\exists m: \quad \vdash_{\mathcal{G}} \mathbb{C}^s_m \wedge U = \alpha \wedge S_m \preceq \alpha$$

by variance of **T_Var**. This is equivalent to

$$\exists m: \quad \vdash_{\mathcal{G}} \mathbb{C}^s_m \wedge S_m \preceq U$$

due to the fact that $\alpha \notin FTV((\mathbb{C}^s_m).S_m)$. However, the last statement is equivalent to (5.237). Thus (5.236) has been proven.

It has been demonstrated that imperative types can be treated consistently in the framework presented in this dissertation. The definitions of the type **T_Var**, behaviors **get** and **set** along with their related primitives given at the beginning of this section were proven to comply with the requirements set forth for type system consistency. It follows that the subject reduction theorem holds for the type system with imperative types introduced in this manner.

5.5 Extensions

In this section, I will describe several extensions to the simple semantics considered above. All these extensions will be considered from the point of view of their impact on the type system.

5.5.1 Errors, nulls, and exceptions

While the typechecking algorithm guarantees the absence of type errors at run-time, other errors may still arise. A practical programming language has to be able to deal with these errors in a consistent and sound manner.

Example 5.1. Consider a primitive function that implements an integer division:

```
associate (T_Integer, T_Integer) -> T_Integer divide : primitiveIntDiv ;
```

Division of some number by zero is certainly type-correct:

```
var T_Integer i,j;
i := 5;
j := 0;
i.divide(j).print;
```

What should be the result of such a program? Apparently, there is no integer number that could meaningfully be printed by the last statement. □

If primitive functions were not allowed, this problem would not arise, since in the absence of primitives every terminating computation is valid as long as the program is typechecked. In real programming, however, the ability of a program (and, ideally, the formal semantics and the type system) to deal with this kind of errors is crucial. There are several possible solutions to this problem. These solutions are not mutually exclusive, but usually the presence of one of them is sufficient.

The notion of a "null value" is a well-established notion in the database community. Surprisingly, in the programming languages community it did not receive nearly as much theoretical attention, even though it is constantly utilized in practical programming. A null value is, in a sense, a common placeholder for missing, insufficient, or incorrect information. Thus, there can be many different null values (e.g. **uninitialized**, **incorrect** or **absent**). In Example 5.1, for example, the division function would return some null value indicating incorrect result. That value will be printed (probably as a string) and the program will proceed normally.

This approach is in fact adopted by the IEEE standard for floating arithmetic [IEE85]. The role of null values is played by such entities as **NaN**. Unfortunately, no such standard exists for integer arithmetics.

From the point of view of the type system, the following question needs to be answered: what are the types of the null values? The simplest consistent approach would be to create a concrete type **T_Null** at the bottom of the type hierarchy and make all null objects to share that type. Then, every function and/or behavior would be able to produce such a value without violating the type discipline. There are, however, certain problems related to this approach. First, the natural semantics has to be extended to deal with null values in functional positions (since **T_Null** is at the bottom of the hierarchy, it is a subtype of all behavior and function types and therefore null values can legally appear where functions and behaviors can). If the imperative types are present, the semantics of extracting a value from a null value posing as a variable object also has to be specified. Finally, since **T_Null** is also a subtype of all product types, the distinction between behaviors that take one argument and the ones taking several arguments is blurred, which requires a significant additional run-time support. For example, consider the following code:

```
behavior T_Point -> (T_Integer, T_Integer) intCoords;
behavior (T_Integer, T_Integer) -> T_Integer plus;
...
var T_Point p;
...
p.intCoords.plus.print;
```

The last statement is supposed to print the sum of the integer coordinates of the given point. However, the variable **p** may contain a null value in which case **p.intCoords** is likely to produce (by default) a single null value rather than a pair of them. From the point of view of typechecking, this is correct. But now, the behavior **plus** is applied (in a type-correct manner!) to a single object instead of a pair. Thus, a run-time system has to be able to deal with situations like these.

The idea of null values and null types can be taken further. Namely, it is possible to introduce several null types, each for its own purpose. For example, one can establish a common subtype of all numeric types **T_NumNull** and make objects such as **NaN** instances of this type. This approach allows one to specify precisely what kind of null values can be expected in a particular place in the program. However, it still requires additional semantics for function/behavior applications if at least one null type is a subtype of these functional types.

Going even further along this path, objects of null types can have some special behaviors defined on them. These behaviors will depend on the type of nulls. These behaviors can be used, for example, to extract some information stored in the null object at the time of its creation. The resulting null objects will then behave much like *exceptions* in most modern programming languages.

Note that the main problems related to the introduction of various null types and values are related to the language semantics and implementation. Type system issues can be neatly solved by placing null types below the types of objects the null values are designed to replace. Thus, the type system can handle not only null values, but also exceptions (or at least a mechanism quite similar to them).

5.5.2 Local variables and dynamic scoping

The only difference between null values with attributes and the traditional exceptions is the lexical scoping of the local variables in the exception processing code. This problem manifests itself in many other (and more important) places as well. Namely, consider the following code:

Example 5.2.

```
a := fun (x) {
  var T_Integer i;
  i := x;
  return fun (y) { return i + y; }
}
...
(a(5))(6).print;
```

□

This code first calls the anonymous function stored in a with the argument 5. The result is a function, that is called with the argument 6. The result of that application is printed. According to the semantics specified in this chapter, this code is valid and should print 11 since the closure always keeps a copy of its environment. In practice, however, local variables are not kept beyond the existence of the stack frame of their function invocation, and therefore an attempt to access i will fail and return a garbage value. The situation can be even worse:

Example 5.3.

```
a := fun (x) {
  var T_Integer i;
  i := x;
  var b := fun (y) { return i + y; };
  i := 2;
  return b;
}
...
(a(5))(6).print;
```

□

According to the specified semantics, this code should print 8 since the last value stored in i is 8. Even though it is possible to implement a language with such semantics (e.g. LISP and some other functional languages), it usually comes at considerable execution and memory costs.

In order to avoid situations like these, it is possible to create a semantics in which all local variable values accessed inside a local function in their scope will be evaluated eagerly rather than lazily. At the same time, assignment to such variables within such functions would be disallowed. Such semantics would correspond to the behavior of the most today's imperative languages.

In order to see how this semantic change can be supported by the type system consider program transformations of the program given in Example 5.2. The following is the standard transformation:

```
T_Var(T_Function) a := newT_VarT_Function;
a.set(fun (x) {
  T_Var(T_Integer) i := newT_VarT_Integer;
  i.set(x);
  return fun (y) { return i.get.plus(y); }
} );

...
(a.get(5))(6).print;
```

that corresponds to the original semantics. The following translation corresponds to the modified semantics:

```
T_Var(T_Function) a := newT_VarT_Function;
a.set(fun (x) {
  T_Var(T_Integer) i := newT_VarT_Integer;
  i.set(x);
  return { let ival = i.get in { fun (y) { return ival.plus(y); }; }; };
} );

...
(a.get(5))(6).print;
```

Here the value of `i` is extracted eagerly. This allows the implementation to discard the variable `i` while the function that refers to its value is still around. Note that this translation automatically ensures that no assignments to `i` are made inside the body of the inner function, as the type of **`ival`** is not a variable type.

Thus it has been shown that the type discipline presented can support both functional (lazy evaluation) and imperative (eager evaluation) styles.

5.5.3 Non-local returns

Another common feature of imperative programming languages that has not yet been addressed is the presence of non-local returns. In the presented semantics, the return expression has to be the last expression in the function body. In imperative programming this restriction is too severe to be practical. The semantic approach to non-local returns is the usage of *continuations*. Typing in the presence of continuations is quite straightforward, however their presence unnecessarily complicates the semantics. Since the primary purpose of this dissertation is the development of the type system, continuations were not formally treated here.

5.5.4 Handling object creation

Object creation/deletion can be handled within the presented framework analogously to the handling of imperative types (Section 5.4). An existential predicate has to be introduced as an additional part of the state and the object creation primitive function semantics has to be specified to change the value of that predicate at a particular point. Once this is done, no changes to the type system are required to deal with object creation. The type system already in place will ensure absence of type errors in a correctly typechecked program.

In this section, several possible semantical extensions and their influence on the type system have been addressed. Next section discusses some key theoretical features of the proposed type system, compares it to other works in the area, and outlines future research direction with respect to further development of the type system theory.

5.6 Discussion

In this section, several additional aspects of the presented type theory will be discussed. Section 5.6.1 describes the modularity of the theory presented so far. Section 5.6.2 discusses the effect that a type system transformation may have on various stages of the type checking. It also uses the above discussion to show how type system evolution can be supported by the presented type system. Section 5.6.3 lists engineering trade-offs related to the usage of the presented type system. Finally, Section 5.6.4 presents a summary of the major features of the presented type system from the theoretical point of view.

5.6.1 Extensibility

One of the significant differences between the theory developed in this chapter and other type theories developed so far is its modularity. The main part of the theory, including the entailment algorithms and the subject reduction theorem, depend only on a very limited set of assumptions about the user type graph and the primitive function semantics. No assumptions at all are made about the *store* except for its initial validity. This property gives the language designer a very flexible and robust system in which almost any store and primitive function semantics can be plugged in with very little effort.

As an example, a type system that includes imperative types and state-preserving objects known to be a challenge for type systems combining parametric and inclusion polymorphism has been fully developed in Section 5.4. The only part of this theory that required some work was the proof of consistency of the primitive functions. If a primitive function does not change the store, the proofs of validity become even simpler.

5.6.2 Type system evolution

So far, the user type graph was considered to be constructed at the beginning of the program analysis process and used later during program typechecking. This assumption might be valid in theoretical settings, but it is quite impractical as big programs are usually built from smaller blocks (libraries) that are typechecked and verified before the complete program is assembled. The question that can be asked of a type system with respect to the above scenario is: what are the program transformations that do not invalidate type checking or invalidate only a part of it?

In this section, various aspects of program verification presented in Section 4.2 will be evaluated from this point of view. Before it is done, an important class of program transformations called *covariant transformations* is introduced.

Covariant transformations include:

1. Adding a new type;
2. Adding a new sub/supertype link (under the condition that the user type graph remains acyclic);
3. Adding a new behavior;
4. Adding a new behavior definition or association to an existing behavior;
5. Replacing an existing behavior definition (association) with a definition (association) with a subtype of the original;
6. Adding a new constant definition;
7. Replacing an existing constant definition with a definition with a subtype of the original;
8. Changing a function in an existing association.

These transformations are called *covariant* because they covariantly change the initial typing environment. They are important because covariant change of the initial typing environment gives each expression the same or lesser type than the type given to that expression in the presence of the original typing environment. In a sense, covariant transformations assure that all objects in the program continue to conform to their original specifications.

Local monotonicity defined in Section 4.2.2 is invariant with respect to covariant transformations in the following sense: if a new definition is added, it has to be checked for local monotonicity, but no old definitions have to be rechecked. This notion of invariance will be used throughout this section in the analysis of various aspects of type consistency.

Existence of a valid ranking for the user type graph defined in Section 4.2.3 is invariant with respect to all covariant transformations except for type and subtype link additions. Type and link additions do not affect existence of a valid ranking as long as the added types and links do not have constraints and the added links do not introduce new paths between types in the original graph. If any of the above conditions are violated, the existence of the valid ranking for the user type graph needs to be rechecked. Note that since existence of the valid ranking is required for algorithm termination, but not for its correctness, in some situations it may be feasible to skip this check as long as there is a safeguard (such as a time constraint) that ensures termination of the type checker.

Constraint consistency as defined in Section 4.2.5 is invariant with respect to covariant transformations. This follows from the properties of entailment discussed later in this section.

Global behavior consistency (Section 4.2.6) describes the conditions placed on the relationships between argument and result types of different definitions given for the same behavior. It is based on entailment and is therefore invariant with respect to covariant transformations.

Functional consistency described in Section 4.2.7 is in a sense the typechecking proper as it checks the conformance of the program to its specification. Functional consistency is also entailment-based and thus invariant with respect to covariant transformations. This is in contrast to ML-style type inference systems where a local change in a function body can potentially affect typing of other functions in the program.

Dispatch consistency conditions (Section 4.2.8) are invariant to all covariant transformations except for subtype/supertype link additions and behavior definition (association) additions and changes. These conditions are the most volatile with respect to program transformations since they are based on the notion of a *concrete set* which is in effect a set of concrete type constructors *below* a given one. Thus, an addition of a subtype can change concrete sets for all its supertypes and thus violate dispatch consistency conditions. The conservative approach to dispatch consistency rechecking is as follows:

1. If a subtype link is added, let S be the set of type constructors that acquire new concrete type constructors as children as a result of this addition. Then, every behavior that has type constructors from S in the primary form of the argument of at least one of its definitions or associations has to be fully rechecked for dispatch consistency. "Full rechecking" here means that *all* definitions and associations for this behavior have to be rechecked.

2. If a behavior definition or association is added or its type is changed for a particular behavior, that behavior has to be fully rechecked.

Thus adding a subtype link at the bottom of the hierarchy leads to full rechecking for a lot of behaviors. This is unfortunate and most probably unnecessary. There are two possible approaches to this problem. The first one is to create more precise rules about rechecking behaviors in this case that would eliminate unnecessary rechecking in simple cases (which are likely to constitute at least 90% of all behavior definitions). The second one is to ignore the potential danger of not rechecking the behaviors thus opening the possibility of "message not understood" and "message ambiguous" errors in the program that is build incrementally. While this last approach clearly defeats the purpose of type checking, it can be practical in prototyping stage of program development when the possibility of run-time errors can be traded for faster development time. In this case the whole program will have to be rechecked at the production stage. Other hybrid approaches are also possible.

Most of the consistency checks described above are invariant with respect to covariant type transformations because they are based on the entailment algorithms of Section 4.3 that produce results invariant with respect to covariant transformations. This follows from Lemma 5.2 and Theorem 5.3.1.

Note that the entailment testing algorithm 4.2 could have been made significantly faster (but still correct) if it was allowed to use the flattened form of the premises (already computed on the first step) for the construction of extended type graph. However, such modification would lead to the loss of invariance with respect to covariant transformations. Consider the following example. Let the following definitions be given:

```
type b(covar X);
type c(covar X);
type d;
(X <= d) c(X) <= b(X);
```

These specify that $c(\alpha) \preceq b(\alpha)$ if $\alpha \preceq d$. Let the entailment to be checked be

$$c(\alpha) \preceq b(\alpha) \vdash \alpha \preceq d \tag{5.238}$$

The current entailment algorithm after verifying that $\vdash c(\alpha) \preceq b(\alpha)$ will attempt to prove that

$$c(a) \preceq b(a) \vdash a \preceq d$$

in a type graph with an additional type constructor a (and an additional subtyping relationship $c(a) \preceq b(a)$). This will fail as even $\vdash a \preceq d$ is not true. Thus the algorithm will conclude that (5.238) is not true.

The modified algorithm, on the other hand, will first infer the flattened form of $c(\alpha) \preceq b(\alpha)$ which is

$$\alpha \preceq \beta \wedge \beta \preceq d \wedge \beta \preceq \alpha$$

which is equivalent to

$$\alpha \preceq d$$

Thus the modified algorithm will attempt to prove

$$\vdash a \preceq d$$

in a type system with an additional type constructor a and an additional subtyping relationship $a \preceq d$. This immediately succeeds, and thus the modified algorithm with conclude that (5.238) is true.

Consider now the following covariant transformation of the original type system:

```
type b(covar X);
type c(covar X);
type d;
(X <= d) c(X) <= b(X);
type e;
e <= b(X);
c(X) <= e;
```

In other words, a new type e was inserted between b and c. Now the entailment (5.238) is false. Indeed, consider $\alpha = e$. Then

$$c(\alpha) = c(e) \preceq e \preceq b(e) = b(\alpha)$$

but

$$e \not\preceq d$$

Since both original and modified algorithms are correct, they return negative answer to (5.238) in this case. Thus, the modified algorithm returns the result different from that returned for the original type system and therefore the modified algorithm is *not* invariant with respect to covariant transformations.

In this section, the behavior of various stages of program verification with respect to covariant program transformations was examined. The conditions under which these stages do not have to be repeated for the modified program have been established. The next section briefly discusses the main engineering trade-offs of the proposed type system.

5.6.3 Engineering trade-offs

The expressive power of this type system comes at a price. First, the proposed type system is significantly more complex than most of the type systems reviewed in Section 2.7. The user is shielded from this increased complexity by the layered design, but a language implementor or database designer would still have to cope with it. System uniformity and orthogonality will reduce the necessary education time, but it may still be quite substantial.

Second, the uniform treatment of all the objects in the system also has its price tag. Since every entity (including numbers, characters, strings etc.) is an object and, therefore, can have behaviors that are dynamically dispatched on its type, the type indicator sufficient for dispatch purposes has to be maintained at run-time. This has an impact on memory requirements.

Third, multiple dispatch utilized in this type system has a negative impact on execution performance. Even in the presence of efficient multi-method dispatch techniques, multiple dispatch is still inherently slower than single dispatch. However, a good optimizing compiler can potentially optimize away most of the necessary dynamic method dispatch by carefully analyzing the context of the call sites or their run-time properties, as demonstrated in [DGC95, DDG+96, HU96].

In the next section, the brief summary of the important features of the theory developed in this chapter will be presented.

5.6.4 Summary

The approach taken in this chapter can be summarized as follows. The program is analyzed to yield a set of *constrained types* and relationships between them. These relationships are translated into sets of subtyping constraints. A program typechecks if these sets admit a solution. This solution does not have to be found, since mere fact of its existence is sufficient to ensure run-time correctness. Thus the algorithms attempt to find inconsistencies in the given constraint sets rather than find their solution. The constraints are interpreted in the domain of *regular trees* equipped with a subtyping relationship and lower and upper bound operators.

The main features of the presented theory are:

1. Inclusion polymorphism (substitutability);
2. Parametric polymorphism;
3. Partial type inference (only behavior types need to be explicitly specified);
4. Provable decidability and correctness;
5. Explicit type constraints;
6. Support for union and intersection types;
7. Modularity (primitive functions and store semantics can be easily modified);
8. Support for imperative types;
9. Support for type system evolution (covariant transformations);
10. The ability to deal with subtyping between types of different variances;
11. Support for distinction between behavior definitions and associations;
12. Dispatch granularity different from the type specification granularity.

This concludes the presentation of the type system theory. In this chapter, decidability and correctness of the algorithms and techniques presented in Chapter 4 has been proven. The effect of various semantical extensions on the theory and the algorithms has been considered. Issues related to the type system evolution and its effect on typechecking were also discussed. Finally, a summary of the major theoretical features of the presented type system has been given. The review of various theoretical type systems has been presented earlier in Section 2.7.1. The next chapter concludes the dissertation and discusses its main contributions as well as directions for future research.

Chapter 6

Conclusions

6.1 Summary and contributions

In this dissertation, a type system for an object-oriented programming language has been developed.

Analysis of theoretical and practical requirements placed on the type system by the language and database components of the system was performed to yield a set of requirements the type system should satisfy. A number of existing and proposed type systems have been examined and evaluated from the point of view of these requirements. This extensive analysis is the first contribution of this dissertation as it covers a very broad spectrum of type systems, yet evaluates them from the same uniform standpoint: the set of requirements obtained by the previous analysis. It is shown that for each required feature there is at least one type system that has it. However, the desired combination of features remains elusive in that no type system, theoretical or practical, existing in a programming language or proposed, has all the features necessary for the consistent and uniform treatment of object-oriented database programming.

Based on these requirements, a type system is designed and various aspects of it are discussed. This feature-rich design of the type system is the second contribution of this work. The design process yields a consistent and uniform type system that satisfies the requirements developed earlier and successfully passes the tests from the type system expressibility test suite. The main design features of the developed type system are consistent combination of parametric and inclusion polymorphism, an elaborate and precise mechanism for behavior (message) typing, handling of precise and complex types of database query operations, uniform integration of imperative features, provisions for schema evolution, and a clean separation between interface, implementation (code), and representation (data layout). The three-layer type system design provides additional flexibility in specification and use of types, and facilitates better code reuse. A major motivation for the layered design is understandability and usability of the type system by programmers. The system can be used at all the three layers; the deeper the layer, the more power it provides, and the greater complexity it involves.

The designed type system is used in a prototype object-oriented programming language (termed *source language* throughout this dissertation). This language is more then just a testbed for the developed type system, it is powerful enough to describe all the major components of object-oriented programming and design, including type and function specification and verification, handling of dynamic and imperative types and objects of these types, multiple dispatch, separation between behavior declarations and definitions on both the type and implementation level, and encoding of database queries via precisely typed behavior applications. This source language can be used as a core of a real object-oriented database programming language as it includes the dispatch procedure for multiple dispatch of behaviors defined in the language. A program written in the source language is verified by first translating it into a more theoretical and simplified *target language* and then applying the typechecking techniques and verification procedures to the resulting program. The main typechecking algorithms (functional consistency checking) are *local* in that a code change inside a function body does not affect the typing or validity of the other functions in the program.

The third contribution of this dissertation is the development of the source language along with the algorithms and procedures for verification and typechecking of the programs written in it.

Finally, a theory has been developed to rigorously prove the validity of the presented typechecking mechanisms. This theory includes a model of types as regular trees, formal treatment of subtyping, constraint types, and entailment. It also includes a natural semantics of the target language designed to formalize and reason about its run-time properties. Both algorithm termination and subject reduction have been proven. The theory was developed in a modular fashion so as to allow its application to other languages and systems. The proof of the validity of the handling of imperative types in the target language was presented as a non-trivial example of utilization of the power inherent in this modular approach. The theory was further examined by looking at the ways it can be extended to support semantic extensions needed for a full-fledged programming language. Another point of discussion was related to the behavior of the presented algorithms in the presence of type system changes. An important class of type system transformations (*covariant transformations*) has been identified and their impact on typechecking and program verification has been carefully examined. The theory has thus been shown to be modular, extensible, and incremental to a certain degree. The developed theory also supports all major type system and design features of the source and target language, including parametric and inclusion polymorphism, imperative, union, and intersection types, type constraints, a very liberal notion of subtyping, different granularity for type and dispatch specifications, and separation between interface (type) and implementation (code). It also features partial type inference that is known to reduce the burden of writing type annotations for every object in a program. At the same time, type inference is localized in order to contain the effect of function code modifications. The development of this type theory and the proofs of decidability and correctness of the typechecking algorithms are the fourth major contribution of this dissertation.

6.2 Future research

While significant work has been done to achieve the primary goal of this dissertation, there is still a considerable room for future research.

On the practical side, making the main verification algorithms more efficient is quite important. While the algorithms presented herein are correct, the main (entailment) algorithm appears to be exponential and thus needs to be improved upon before it can be used in a practical language. Implementation of the source and target languages as well as an efficient implementation of translation and verification algorithms remain to be accomplished. Another interesting and promising research venue is related to the development of an efficient dispatch strategy at a finer granularity. In this dissertation, the dispatch on parametric types does not depend on type parameter. This is done to utilize the existing efficient multi-method dispatch techniques. However, if an efficient mechanism capable of taking into account type parameters can be developed, it will greatly simplify the theory and make the language more powerful. Finally, development of a database programming language based on this type system presents a whole new set of challenges, such as the semantics of persistence, handling of persistent types, behaviors, and functions, and efficient implementation of database queries and updates.

From the design prospective, addition of exceptions into the language seems to be an interesting and potentially rewarding research direction. Another design possibility worth pursuing is the addition of a module system to the source language. From the database prospective, adding transaction support directly into the language seems to be a promising idea. Yet another interesting research direction is incorporation of *algebraic types* (type constructors that serve also as value constructors, as in Standard ML [MTH90]) into the type system.

From the theoretical point of view, there are a number of issues that warrant further study. One of the most important theoretical research directions is a study into the possibility of lifting or significantly relaxing the valid ranking requirement placed on the user type graph. The author believes it can be accomplished without sacrificing decidability. Another theoretical research possibility is research into the complexity of the presented algorithms and its theoretical limits. Finally, development of the full-fledged language semantics (with continuations) and the necessary modification of the relevant proofs is a challenging task.

The last task is just one of the integral parts of a larger problem of developing a powerful, theoretically sound, efficient, and practical object-oriented database programming language for the next millennium. I believe such a development will be undertaken, and the result will probably exceed the expectations of both research and industrial communities. My hope is that the research presented in this dissertation is a step along this long and difficult road.

Bibliography

[AB87] M. P. Atkinson and O. P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[ABD+92] M. Atkinson, F. Banchilon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In F. Banchilon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, 1992.

[AC93] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.

[ACO85] A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly-typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[Ada95] *Ada 95 Reference Manual*, 1995. Available electronically.
URL: `http://www.adahome.com/rm95`

[ADL91] R. Agrawal, L. G. DeMichiel, and B. G. Lindsay. Static type checking of multi-methods. *ACM SIGPLAN Notices*, 26(11):113–128, November 1991. In Proc. of OOPSLA'91.

[AFM97] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parametrization to the Java language. In *Proceedings of the OOPSLA'97*, 1997.

[AG89] R. Agrawal and N. H. Gehani. ODE (object database and environment): The language and the data model. In *Proceedings of the ACM-SIGMOD 1989 International Conference on Management of Data*, pages 36–45, 1989.

[AG96] K. Arnold and J. Gosling. *The Java Language Specification.* Addison-Wesley, 4th edition, 1996. ISBN 0-201-63455-4.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *VLDB Journal*, 4:403–444, 1995.

[AM95] M. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[AN80] A. Arnold and M. Nivat. The metric space of infinite trees. Algebraic and topological properties. *Fundamenta Informaticae III*, pages 445–476, 1980.

[App92] A. W. Appel. A critique of standard ML. Technical Report CS-TR-364-92, Princeton University, November 1992.

[App94] Apple Computer, Inc. *Dylan Interim Reference Manual*, 1994.

[AVWW96] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang.* Prentice Hall, 2nd edition, 1996.

[AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. Technical Report RJ 9454 (83075), IBM Research Division, August 1993.

[AWL94] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Conference Record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 163–173. ACM Press, 1994.

[BBdB+93] R. Bal, H. Balsters, R. A. de By, A. Bosschaart, J. Flokstra, M. V. Keulen, J. Skowronek, and B. Termorshuizen. *The TM Manual*, December 1993. Version 2.0 revision C. Available electronically.
URL: `ftp://ftp.cs.utwente.nl/pub/doc/TM`

[BC95] J. Boyland and G. Castagna. Type-safe compilation of covariant specialization: A practical case. Technical Report UCB/CSD-95-890, University of California, Computer Science Division (EECS), Berkeley, California 94720, November 1995.

[BC97] J. Boyland and G. Castagna. Parasitic methods: An implementation of multi-methods in Java. *SIGPLAN Notices*, 32(10):66–76, 1997. Proceedings of OOPSLA'97.
URL: `ftp://ftp.ens.fr/pub/dmi/users/castagna/oopsla97.ps.gz`

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[BDG+88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification, June 1988. X3J13 Document 88-002R.

[BDMN79] G. M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur (Lund, Sweden), Bratt Institute Fuer Neues Lerned (Goch, FRG), Chartwell-Bratt Ltd (Kent, England), 1979.

[BFP96] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *Informal Proceedings of The Fourth Workshop on Foundations of Object-Oriented Languages (FOOL 4)*, 1996. Contributed talk.

[BG93] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications*, 1993.

[BG96] M. F. Barrett and M. E. Giguere. A note on covariance and contravariance unification. *ACM SIGPLAN Notices*, 31(1):32–35, January 1996.

[BLR96] G. Baumgartner, K. Läufer, and V. F. Russo. Interaction of object-oriented design patterns and programming languages. Technical Report CSD-TR-96-020, Deptartment of Computer Sciences, Purdue University, 1996.

[BM96a] F. Bourdoncle and S. Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of the 24th ACM Conference on Principles of Programming Languages (POPL'24)*, 1996.

[BM96b] F. Bourdoncle and S. Merz. Primitive subtyping $\vee$ implicit polymorphism $\vdash$ object-orientation. In *Foundations of Object-Oriented Languages 3*, 1996. Extended abstract.

[BO96] P. Buneman and A. Ohori. Polymorphism and type inference in database programming. *ACM Transactions on Database Systems*, 21(1):30–76, March 1996.

[BOSW98a] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ: Extending the Java programming language with type parameters. Manuscript, March 1998. Revised August 1998.

[BOSW98b] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. GJ specification. Manuscript, May 1998.

[BOSW98c] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Proceedings of the 13th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 183–200, October 1998.

[BOW98] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings of the 1998 European Conference on Object-Oriented Programming (ECOOP'98)*, 1998.

[BP94] A. Black and J. Palsberg. Foundations of object-oriented languages. *ACM SIGPLAN Notices*, 29(3):3–11, 1994. Workshop Report.

[BP99] P. Buneman and B. Pierce. Union types for semistructured data. Technical Report MS-CIS-99-09, Department of CIS, University of Pennsylvania, 1999.

[Bra92] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity And Multiple Inheritance*. PhD thesis, University of Utah, Department of Computer Science, March 1992.

[Bru96] K. K. Bruce. Typing in object-oriented languages: Achieving expressibility and safety, 1996.
URL: `ftp://cs.williams.edu/pub/Kim/Static.ps`

[BSG94] K. B. Bruce, A. Schuett, and R. V. Gent. A type-safe polymorphic object-oriented language. Accessible by anonymous FTP, July 1994.
URL: `ftp://cs.williams.edu/pub/kim/PolyTOIL.dvi`

[BSG95] K. B. Bruce, A. Schuett, and R. V. Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, Åarhus, Denmark, August 1995. LNCS 952. Springer. Extended abstract.

[Car86a] L. Cardelli. Amber. In G. Cousineau, P.-L. Curien, and B. Robinet, editors, *Combinators and Functional Programming Languages*. Springer-Verlag, 1986. LNCS 242.

[Car86b] L. Cardelli. A polymorphic λ-calculus with **Type:Type**. Technical Report 10, DEC SRC, 130 Lytton Avenue, Palo Alto, CA 94301, May 1986. SRC Research Report.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[Car89] L. Cardelli. Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, IFIP State of the Art Reports Series. Springer-Verlag, February 1989.
URL: `http://www.luca.demon.co.uk/Bibliography.html`

[Car93] L. Cardelli. An implementation of $F_{<:}$. Technical Report 97, DEC Systems Research Center, February 1993.

[Car97] L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103. CRC Press, 1997.
URL: `http://www.luca.demon.co.uk/Bibliography.html`

[Cas95a] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[Cas95b] G. Castagna. A meta-language for typed object-oriented languages. *Theoretical Computer Science*, 151(2):297–352, November 1995.

[Cas96] G. Castagna. *Object-Oriented Programming: A Unified Foundation*, chapter Type Systems for Object-Oriented Programming. Progress in Theoretical Computer Science. Birkaüzer, Boston, 1996.

[CBB+97] R. G. G. Cattell, D. Barry, D. Bartels, M. Berler, J. Eastman, S. Gamerman, D. Jordan, A. Springer, H. Strickland, and D. Wade. *The Object Database Standard: ODMG 2.0.* Morgan Kaufmann Publishers, Los Altos (CA), USA, 1997.

[CCH+89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the ACM Conference on Functional Programming and Computer Architecture*, pages 273–280, 1989.

[CGL95] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, February 1995.

[Cha92a] C. Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, March 1992.

[Cha92b] C. Chambers. Object-oriented multi-methods in Cecil. In O. L. Madsen, editor, *ECOOP '92, European Conference on Object-Oriented Programming, Utrecht, The Netherlands*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56. Springer-Verlag, New York, N.Y., 1992.

[Cha93] C. Chambers. The Cecil language: Specification and rationale. Technical Report TR 93-03-05, Department of Computer Science and Engineering, FR-35, University of Washington, March 1993.

[CL95] C. Chambers and G. T. Leavens. Typechecking and modules for multimethods. *ACM Transactions on Programming Languages and Systems*, 17(6):805–843, November 1995.

[CL96] C. Chambers and G. T. Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. Technical Report 96-17, Department of Computer Science, Iowa State University, December 1996.

[CL97] C. Chambers and G. T. Leavens. BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing. In *FOOL 4, The Fourth International Workshop on Foundations of Object-Oriented Languages, Paris, France*, January 1997.

[CMM91] R. C. H. Connor, D. J. McNally, and R. Morrison. Subtyping and assignment in database programming languages. In *Proceedings of the 3rd International Workshop on Database Programming Languages*, Napfilon, Greece, 1991.

[CMMS91] L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of system F with subtyping. In T. Ito and A. R. Meyer, editors, *International Conference on Theoretical Aspects of Computer Software*, pages 750–770, 1991. Lecture Notes in Computer Science 526.

[CO94] K. Chen and M. Odersky. A type system for a lambda calculus with assignment. In *Proc. Theoretical Aspects of Computer Science, Sendai, Japan*, April 1994. Spinger LNCS.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[Cou86] B. Courcelle. Equivalence and transformation of regular systems — applications to recursive program schemes and grammars. *Theoretical Computer Science*, 42:1–122, 1986.

[CT95] W. Chen and V. Turau. Multiple dispatching based on automata. *Journal of Theory and Practice of Object Systems*, 1(1), 1995.

[CTK94] W. Chen, V. Turau, and W. Klas. Efficient dynamic look-up strategy for multi-methods. In M. Tokoro and R. Pareschi, editors, *ECOOP '94, European Conference on Object-Oriented Programming*, volume 821 of *Lecture Notes in Computer Science*, pages 408–431, New York, N.Y., July 1994. Springer-Verlag.

[DAS96] E. Dujardin, E. Amiel, and E. Simon. Fast algorithms for compressed multi-method dispatch tables generation. *ACM Transactions on Programming Languages and Systems*, 1996.

[DCG94] J. Dean, C. Chambers, and D. Grove. Identifying profitable specialization in object-oriented languages. Technical Report 94-02-05, Department of Computer Science and Engineering, FR-35, University of Washington, February 1994.

[DDG+96] J. Dean, G. DeFouw, D. Grove, V. Livinov, and C. Chambers. Vortex: an optimizing compiler for object-oriented languages. *ACM SIGPLAN Notices*, 31(10):83–100, October 1996.

[DE97] S. Drossopoulou and S. Eisenbach. Java is type safe — probably. In *Proceedings of the 11th European Conference on Object Oriented Programming (ECOOP'97)*, June 1997.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In W. Olthoff, editor, *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, Åarhus, Denmark, August 1995. LNCS 952. Springer.

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs where clauses: Constraining parametric polymorphism. *SIGPLAN Notices*, 30(10):156–168, October 1995.

[EST95a] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. *SIGPLAN Notices*, 30(10):169–184, October 1995.

[EST95b] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. *Electronic Notes in Theoretical Computer Science*, 1, 1995. URL: `http://www.elsevier.nl/locate/entcs/volume1.html`

[ESTZ95] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *LISP and Symbolic Computation*, 8(4):357–397, 1995.

[FM96] K. Fisher and J. C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996. URL: `ftp://theory.stanford.edu/pub/jcm/papers/tapos.ps`

[Fra97] M. Franz. The programming language Lagoona — A fresh look at object-orientation. *Software — Concepts and Tools*, 18:14–26, 1997.

[Ghe91] G. Ghelli. A static type system for message passing. *ACM SIGPLAN Notices*, 26(11):129–145, November 1991. In Proc. of OOPSLA'91.

[GM96] A. Gawecki and F. Matthes. Integrating subtyping, matching and type quantification: A practical perspective. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, Linz, Austria, July 1996. Springer Verlag.

[GR89] A. Goldberg and D. Robson. *ST-80, The Language*. Addison-Wesley, 1989.

[Har92] S. P. Harbison. *Modula-3*. Prentice Hall, 1992.

[Hau93] F. J. Hauck. Towards the implementation of a uniform object model. In A. Bode and M. D. Cin, editors, *Parallel Computer Architectures: Theory, Hardware, Software, and Applications - SFB Colloquium SFB 182 and SFB 342*, number 732 in Lecture Notes in Computer Science, pages 180–189. Springer, October 1993.

[HJW+92] P. Hudak, S. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. *Report on the Programming Language Haskell*, March 1992. Version 1.2. Available electronically.
URL: **http://www.haskell.org/definition/haskell-report-1.2.ps.gz**

[HLPS99] W. Holst, Y. Leontiev, C. Pang, and D. Szafron. Multi-method dispatch using product-type tries. Technical Report TR99-01, University of Alberta, 1999.

[Hol93a] W. Holst. Modular Smalltalk: A first implementation. Technical Report TR 93-07, Department of Computing Science, University of Alberta, 1993.

[Höl93b] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *Proceedings of ECOOP'93*, 1993.

[HSPL98] W. Holst, D. Szafron, C. Pang, and Y. Leontiev. Multi-method dispatch using single-receiver projections. Technical Report TR98-03, University of Alberta, 1998.

[HU96] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object oriented languages. *ACM Transactions on Programming Languages and Systems*, 18(4):355–400, July 1996.

[IEE85] IEEE standard for binary floating-point arithmetic. ANSI/IEEE Standard 754-1985, Institute of Electrical and Electronics Engineers, August 1985.

[JW75] K. Jensen and N. Wirth. *The Programming Language Pascal*. Springer-Verlag, 1975.

[KCMS96] G. N. C. Kirby, R. C. H. Connor, R. Morrison, and D. Stemple. Using reflection to support type-safe evolution in persistent systems. Technical Report CS/96/10, University of St Andrews, 1996.

[Kim93] W. Kim. Object-oriented database systems: Promises, reality, and future. In *Proceedings of the 19th VLDB Conference*, pages 676–687, 1993.

[KT92] W. Klas and V. Turau. Persistence in the object-oriented database programming language VML. Technical Report TR-92-045, International Computer Science Institute, 1947 Center St., Suite 600, Berkeley, CA 94704-1198, July 1992.

[LCD+95] B. Liskov, D. Curtis, M. Day, S. Ghemawat, R. Gruber, P. Johnson, and A. C. Meyers. *Theta Reference Manual*. MIT Laboratory for Computer Science, Cambridge, MA 02139, February 1995.
URL: **http://www.pmg.lcs.mit.edu/papers/thetaref/**

[Lit98] V. Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Proceedings of the 1998 Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'98)*, 1998.

[LM98] G. T. Leavens and T. D. Millstein. Multiple dispatch as dispatch on tuples. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 374–387, October 1998.
URL: **http://www.cs.washington.edu/homes/todd/papers/oopsla98.ps**

[LML$^+$94] V. G. Lutiy, A. B. Merkov, Y. V. Leontiev, E. J. Gawrilow, N. A. Ivanova, M. E. Iofinova, M. L. Paklin, and A. K. Hodataev. *DBMS Modula-90K*. RAN Data Processing Center, Moscow, 1994. /in Russian: Sistema Programmirovaniya Baz Dannyh Modula-90K/.

[LÖS98] Y. Leontiev, M. T. Özsu, and D. Szafron. On separation between interface, implementation, and representation in object DBMSs. In *Proceedings of TOOLS-26'98*, Santa Barbara, California, August 1998.

[LP91] W. R. LaLonde and J. Pugh. Subclassing $\neq$ subtyping $\neq$ is-a. *Journal of Object-Oriented Programming*, 3(5):57–62, January 1991.

[LRV92] C. Lécluse, P. Richard, and F. Vélez. O_2, an object-oriented data model. In F. Banchilon, C. Delobel, and P. Kanellakis, editors, *Building an Object-Oriented Database System: The Story of O_2*, 1992.

[LW94] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[LW95] G. T. Leavens and W. E. Weihl. Specification and verification of object oriented programs using supertype abstraction. *ACTA Informatica*, 32(8):705–778, November 1995.

[MBC$^+$96] R. Morrison, F. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. *Napier88 Reference Manual*. University of St. Andrews, March 1996. Release 2.2.1.

[MBL97] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM Simposium on Principles of Programming Languages (POPL'97)*, January 1997.

[Mey88] B. Meyer. *Eiffel — the language*. Prentice-Hall, 1988.

[MHH91] W. B. Mugridge, J. Hamer, and J. G. Hosking. Multi-methods in a statically-typed programming language. In P. America, editor, *ECOOP'91*, pages 307–324, Geneva, Switzerland, July 1991. LNCS 512.

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993. ISBN 0-201-62430-3.

[MMS94] F. Matthes, S. Müßig, and J. W. Schmidt. Persistent polymorphic programming in Tycoon: An introduction. FIDE Technical Report Series FIDE/94/106, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.

[MS91] F. Matthes and J. Schmidt. Bulk types: Built-in or add-on? In *Proceedings of the Third International Workshop on Database Programming Languages*. Morgan Kaufmann Publishers, September 1991.

[MS92] F. Matthes and J. Schmidt. Definition of the Tycoon language — a preliminary report. Technical Report FBI-HH-B-160/92, Universität Hamburg, October 1992.

[MTH90] R. Miller, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[MW93] H. Mössenbök and N. Wirth. The programming language Oberon-2. Manuscript, October 1993. Institut für Computersysteme, ETH Zürich.

[MW97] S. Marlow and P. Wadler. A practical subtyping system for Erlang. In *Proceedings of the Second International Conference on Functional Programming*, Amsterdam, June 1997.

[NP91] O. Nierstrasz and M. Papathomas. Towards a type theory for active objects. *ACM OOPS Messenger*, 2(2):89–93, April 1991. Proceedings of the OOPSLA/ECOOP'90 Workshop on Object-Based Concurrent Systems.

[OBBT89] A. Ohori, P. Buneman, and V. Breazu-Tannen. Database programming in Machiavelli — a polymorphic language with static type inference. *SIGMOD Record*, 18(2):46–57, 1989.

[OL96] M. Odersky and K. Läufer. Putting type annotations to work. In *Proc. 23rd ACM Symposium on Principles of Programming Languages*, pages 65–67, January 1996.

[ÖPS$^+$95] M. T. Özsu, R. J. Peters, D. Szafron, B. Irani, A. Lipka, and A. Muñoz. TIGUKAT: A uniform behavioral objectbase management system. *The VLDB Journal*, 4(3):445–492, July 1995.

[OW97] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Simposium on Principles of Programming Languages (POPL'97)*, January 1997.

[Pet94] R. J. Peters. TIGUKAT: A uniform behavioral objectbase management system. Technical Report TR 94-06, Department of Computing Science, University of Alberta, April 1994.

[PHLS99] C. Pang, W. Holst, Y. Leontiev, and D. Szafron. Multi-method dispatch using multiple row displacement. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, 1999.

[Pot96] F. Pottier. Simplifying subtyping constraints. *ACM SIGPLAN Notices*, 31(6):122–133, June 1996.

[Pot98] F. Pottier. *Type Inference in the Presence of Subtyping: from Theory to Practice*. PhD thesis, Université Paris VII, April 1998.

[PS94] J. Palsberg and M. I. Schwartzbach. *Object-Oriented Type Systems*. John Wiley & Sons, 1994.

[QKB96] Z. Qian and B. Krieg-Brueckner. Typed OO functional programming with late binding. In P. Cointe, editor, *Proceedings of the 10th European Conference on Object-Oriented Programming*, volume 1098 of *LNCS*, pages 48–72. Springer, July 1996.

[RCS93] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 15(3):494–534, July 1993.

[Reh98] J. Rehof. *The Complexity of Simple Subtyping Systems*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, 1998.

[RS97] P. Roe and C. Szyperski. Lightweight parametric polymorphism for Oberon. In *Proceedings of the Joint Modular Languages Conference*, 1997.

[RTL$^+$91] R. K. Raj, E. Tempero, H. M. Levy, A. P. Black, N. C. Hutchinson, and E. Jul. Emerald: A general-purpose programming language. *Software Practice and Experience*, 21(1):91–118, January 1991.

[RW92] M. Reiser and N. Wirth. *Programming in Oberon — Steps beyond Pascal and Modula*. Addison-Wesley, 1992.

[Sar97] V. Saraswat. Java is not type-safe. Available electronically, 1997.
URL: `http://www.research.att.com/~vj/bug.html`

[Seq98] D. Sequeira. *Type Inference with Bounded Quantification*. PhD thesis, Department of Computer Science, University of Edinburgh, 1998. Also Technical Report ECS–LFCS–98–403.

[Sha97] D. Shang. *Transframe: The Annotated Reference.* Software Systems Research Laboratory, Motorola, Inc., Schaumburg, Illinois, January 1997. Draft 1.4.

[SM94] J. W. Schmidt and F. Matthes. The DBPL project: Advances in modular database programming. *Information Systems*, 19(2):121–140, 1994.

[SO96] D. Stoutamire and S. Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute at Berkeley, August 1996.

[Str91] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1991.

[Tai96] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):439–479, September 1996.

[Tho97] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[TNG92] D. Tsichritzis, O. Nierstrasz, and S. Gibbs. Beyond objects: Objects. *International Journal of Intelligent and Cooperative Information Systems*, 1(1):43–60, March 1992.

[TS96] V. Trifonov and S. Smith. Subtyping constrained types. In *Proceedings of the 3rd International Static Analysis Symposium*, pages 349–365, May 1996. Lecture Notes in Computer Science 1145.

[TT94] L. Thorup and M. Tofte. Object-oriented programming and Standard ML. In J. H. Reppy, editor, *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications, Orlando, Florida*, number 2265 in Rapport de recherche, pages 41–49. INRIA, June 1994.

[Tur37] A. M. Turing. Computability and λ-definability. *Journal of Symbolic Logic*, 2:153–163, 1937.

[Wir83] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 2nd edition, 1983.

[Wri93] A. K. Wright. Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Department of Computer Science, Rice University, February 1993.

Appendix A

The Basic TIGUKAT Type System

This chapter gives a full description of the basic TIGUKAT type system. Classes and implementation types are not fully described as their details are implementation-specific.

A.1 Common behaviors

```
behavior typeOf(X)     : T_ConstrainedType;     // Object type
behavior classOf(X)    : T_InfiniteClass(X);    // Object class
behavior itypeOf(X)    : T_ImplementationType; // Object implementation type

behavior OIDequal(X,X) : T_Boolean;      // Object identity equality
behavior equal(X x,X y): T_Boolean       // Equality
                                         // Shortcut: operator binary "=="
        implementation { return x.OIDequal(y); };
notEqual(X x,X y)      : T_Boolean       // Inequality
                                         // Shortcut: operator binary "!="
        implementation { return not (x == y); };

print(X,OutputStream(X));                // Print to an output stream

apply(X, (X):Y) : Y;                     // Function application
                                         // Shortcuts: operator binary ".";
                                         //            juxtaposition
```

A.2 Product types

```
type T_Unit;

type T_ProductN(covar X1, covar X2, ..., covar XN) {
    project1()       : X1;
    project2()       : X2;
    ...
    projectN()       : XN;
};
```

for each N greater than 1.

A.3 Functional types

```
type T_Function(contravar A, covar R) {
    nargs()        : T_Natural;     // number of arguments
    code()         : T_String;      // function code (may be empty)
};

class C_Function implements T_Function(A, R) { ... };

type T_Behavior(contravar A, covar R) subtype of  T_Function(A, R) {
    name()         : T_String;                         // behavior name
    description()  : T_String;                         // Documentation
    definitions()  : T_ConstrainedType;                // all definitions
    associations(): T_Set(T_BehaviorAssociation(A,R)); // all associations
    dispatch(T_List(T_Type)) : T_Function(A, R);       // dispatch
};

class C_Behavior implements T_Behavior(A, R) { ... };
```

The type `T_Function(A,R)` can be abbreviated as `(A):R` or as `A -> R`. The following is the type necessary for behavior type definition:

```
type T_BehaviorAssociation(contravar A, covar R) subtype of T_Object {
    type()        : T_SimpleConstrainedType;       // type
    function()    : T_Function(A,R);               // associated function
};

class C_BehaviorAssociation implements T_BehaviorAssociation(A,R) { ... };
```

The following is the type of low-level, implementation functions

```
type T_ImplementationFunction subtype of T_Object {
    description(): T_String;          // Documentation
    declaration(): T_String;          // host language declaration
    definition() : T_String;          // host language definition
    argtype()    : T_List(T_ImplementationType);// required implementation types
                                                 // of all arguments
};

class C_ImplementationFunction implements T_ImplementationFunction { ... };
```

A.4 Object types

A.4.1 Special types

```
type T_Object;

type T_Null subtype of <all object types>;

class C_Null implements T_Null { ... };
```

A.4.2 Atomic types

```
type T_Discrete subtype of T_Object {
    pred(): selftype;               // Return the previous element
    succ(): selftype;               // Return the next element
};

type T_PartiallyOrdered(contravar X) subtype of T_Object {
    less(selftype) : T_Boolean;           // Antisymmetric comparison.
                                          // Shortcut: operator binary '<'
    greater(selftype x) : T_Boolean       // Antisymmetric comparison.
                                          // Shortcut: operator binary '>'
        implementation { return x < self; };
    lessOrEqual(selftype x) : T_Boolean   // Symmetric comparison.
                                          // Shortcut: operator binary '<='
        implementation { return x < self or x == self; };
    greaterOrEqual(selftype x) : T_Boolean // Symmetric comparison.
                                          // Shortcut: operator binary '>='
        implementation { return x <= self; };
};

type T_Ordered(contravar X) subtype of T_PartiallyOrdered(X) {
    max(selftype x) : selftype            // Maximum of the two
        implementation {
           return if x < self then self; else x; endif };
    min(selftype x) : selftype            // Minimum of the two
        implementation {
           return if x < self then x; else self; endif };
};

type T_Numeric subtype of T_Ordered(T_Numeric) {
    abs() : T_Numeric;              // Absolute value
    negate() : T_Numeric;           // Negation.
                                    // Shortcut: operator unary '-'
    add(T_Numeric) : T_Numeric;     // Addition.
                                    // Shortcut: operator binary '+'
    subtract(T_Numeric) : T_Numeric;// Subtraction.
                                    // Shortcut: operator binary '-'
    multiply(T_Numeric) : T_Numeric;// Multiplication.
                                    // Shortcut: operator binary '*'
    divide(T_Numeric) : T_Numeric;  // Division.
                                    // Shortcut: operator binary '/'
};

type T_Boolean subtype of T_Object {
    not() : T_Boolean;               // Negation.
    or(T_Boolean) : T_Boolean;       // Logical OR.
                                     // Shortcut: operator binary 'or'
    and(T_Boolean) : T_Boolean;      // Logical AND.
                                     // Shortcut: operator binary 'and'
    xor(T_Boolean) : T_Boolean;      // Logical XOR.
                                     // Shortcut: operator binary 'xor'
    if(X, X)       : X;              // If-expression. Shortcut:
                                     // 'if ... then ... else ... endif'
};

finite class C_Boolean implements T_Boolean {
    implementation type short atomic int;
    not() implementation function
        not(int self) { return ~self; };
    or(C_Boolean) implementation function
        and(int self, int x) { return self || x; };
    ...
};

type T_Character subtype of T_Discrete, T_Ordered(T_Character) {
    ord  : T_Natural;               // Returns the ordinal value
};

finite class C_ASCIICharacter implements T_Character {
    implementation type short atomic char;
    ...
};

finite class C_UnicodeCharacter implements T_Character {
    implementation type short atomic Unichar;
    ...
};

type T_Real subtype of T_Numeric {
    truncate(): T_Integer;          // Truncate throwing away
                                    // the fractional part
    round()  : T_Integer;           // Round to the nearest
    sign()   : T_Integer;           // Sign: 1 if positive, -1 if
                                    // negative, 0 otherwise
    // Refined from T_Numeric
    abs() : T_Real;
    negate() : T_Real;
    add(T_Real) : T_Real;
    subtract(T_Real) : T_Real;
    multiply(T_Real) : T_Real;
    divide(T_Real) : T_Real;
};

infinite class C_Real implements T_Real {
    implementation type short atomic float;
    ...
};

infinite class C_LongReal implements T_Real {
    implementation type long atomic double;
    ...
};

type T_Integer subtype of T_Real, T_Discrete {
    div(T_Integer) : T_Integer;         // Integer division

    // Refined from T_Real
    abs() : T_Natural;
    negate() : T_Integer;
    add(T_Integer) : T_Integer;
    subtract(T_Integer) : T_Integer;
    multiply(T_Integer) : T_Integer;
};

infinite class C_Integer implements T_Integer {
    implementation type short atomic int;
    ...
};

infinite class C_LongInteger implements T_Integer {
    implementation type long atomic long int;
    ...
};

type T_Natural subtype of T_Integer {
    mod(T_Natural) : T_Natural;           // Remainder

    // Refined from T_Integer
    truncate() : T_Natural;
    round() : T_Natural;
    sign() : T_Natural;
    div(T_Natural) : T_Natural;
    add(T_Natural) : T_Natural;
    multiply(T_Natural) : T_Natural;
};

infinite class C_Natural implements T_Natural {
    implementation type short atomic unsigned;
    ...
};

infinite class C_LongNatural implements T_Natural {
    implementation type long atomic long unsigned;
    ...
};
```

A.4.3 Collection types

```
type T_Collection(covar X) subtype of T_Object {
    hasElement(X) : T_Boolean;       // Checks if the element
                                     // is in the collection
    cardinality() : T_Natural;       // Collection cardinality
    pick()        : X;               // Pick an element
    map((X):Y)    : T_Collection(Y); // Apply the given function to all
                                     // elements and return the
                                     // collection of results
    filter((X):T_Boolean) : T_Collection(X); // Filters the receiver
                                     // collection using the
                                     // argument function as a filter
    sort() where (X subtype of T_Ordered(X)) : T_List(X); // Sorts
                                     // the elements of the
                                     // collection. Only applicable to
                                     // collections of ordered elements
    sort((X,X) : T_Boolean) : T_List(X);// Sorts the collection using
                                     // the supplied sorting function
};


type T_Set(covar X) subtype of T_Collection(X) {
    subset(T_Set(Y))     : T_Boolean;       // Is the receiver a subset
                                            //            of the argument?
    union(T_Set(Y))      : T_Set(lub(X,Y)); // Set union
    intersect(T_Set(Y))  : T_Set(glb(X,Y)); // Set intersection
    difference(T_Set(Y)): T_Set(X);         // Set difference
    addElement(X)        : T_Set(X);        // Add an element
    removeElement(Y)     : T_Set(X);        // Remove an element
    // Refined from T_Collection
    map((X):Y)     : T_Set(Y);
    filter((X):T_Boolean) : T_Set(X);
};

infinite class C_Set implements T_Set(X) {
    const T_List(X) _list;
    ...
};

type T_EmptySet subtype of T_Set(X);

finite class C_EmptySet implements T_EmptySet { ... };

type T_USet(novar X) subtype of T_Set(X) {
    union():=(T_Set(X));       // Destructive set union
    intersect():=(T_Set(Y));   // Destructive set intersection
    difference():=(T_Set(Y)); // Destructive difference
    addElement():=(X);         // Destructively add an element
    removeElement():=(Y)   : T_Boolean; // Destructively remove an element.
                                        // Returns if it was removed
    removeAll();           // Remove all elements
    map:=((X):X);          // Replace all objects in the set by their images
                           // obtained by application of the given function
};

manual class C_USet implements T_USet(X) {
    const T_Array(X) _array;
    ...
};

type T_List(covar X) subtype of T_Collection(X) {
    at(T_Natural)       : X;                  // Get at arg's position
    cat(X)              : T_List(X);          // Concatenation
    slice(T_Natural, T_Natural) : T_List(X); // Slice from i-th
                                              // to j-th element
    // Refined from T_Collection
    map((X):Y)     : T_List(Y);
    filter((X):T_Boolean) : T_List(X);
};

infinite class C_List implements T_List(X) { ... };
```

```
type T_EmptyList subtype of T_List(X);

finite class C_EmptyList implements T_EmptyList { ... };

type T_Array(novar X) subtype of T_List(X) {
    // Refined from T_List
    at(T_Natural)        := X;                   // Set at arg's position
    slice(T_Natural, T_Natural) := T_List(X); // Slice set
};

manual class C_Array implements T_Array(X) { ... };

type T_String subtype of T_List(T_Character) {
    // Refined from T_List
    cat(T_String)       : T_String;
    slice(T_Natural, T_Natural) : T_String;
    filter((T_Character):T_Boolean) : T_String;
};

infinite class C_String implements T_String { ... };

infinite class C_UnicodeString implements T_String { ... };
```

A.4.4 Imperative types

```
type T_Var(novar X) subtype of T_Object {
    get()          : X;                // Get the value
    set(X);                            // Set the value
    destructive((X):X);                // See the note below
};

manual class C_Var(X) implements T_Var(X) { ... };
```

The behavior **destructive** is designed to replace the value stored in the variable by the value generated from the old contents of that variable by the function passed in as an argument. In other words,

```
v.destructive(f)
```

is equivalent to

```
v.set(v.get.f)
```

For example,

```
Integer v;
v := 5;
v.destructive(negate);
v.print(stdout);
```

will print −5 while

```
Integer v;
v := 5;
v.destructive(fun (x) { return x.plus(5); });
v.print(stdout);
```

will print 10.

A.4.5 Metatypes

```
type T_ConstrainedType subtype of T_PartiallyOrdered(T_ConstrainedType) {
    components()     : T_Set(T_SimpleConstrainedType); // Components of this type
};

type T_SimpleConstrainedType subtype of T_ConstrainedType {
    constraints()    : T_Set(T_TypeConstraint);// Constraints placed on this type
    variables()      : T_Set(T_TypeVariable);  // Set of type variables
    root()           : T_OpenTypeOrTV;         // Root of the type tree
};

type T_Type subtype of T_SimpleConstrainedType {
    name()           : T_String;        // The name
    description()    : T_String;        // Documentation
    numParameters()  : T_Natural;       // Number of type parameters
    var(T_Natural n) : T_Variance;      // Variance of the n-th type parameter
    classes()        : T_Set(T_Class);  // All associated classes
    subtypes()       : T_Set(T_Type);   // Primary functors of immediate subtypes
    supertypes()     : T_Set(T_Type);   // Primary functors of
                                        //    immediate supertypes
    subtypings()     : T_Set(T_TypeConstraint); // All subtype definitions
                                        //  involving this type
    deepExtent()     : T_Set(T_Object); // All enumerable objects of this
                                        // type and its subtypes
};

class C_Type implements T_Type { ... };
infinite class C_ConstrainedType implements T_ConstrainedType { ... };
infinite class C_SimpleConstrainedType implements T_SimpleConstrainedType { ... };
```

The partial order on types is subtyping.

The users only define types of the type T_Type that corresponds to a type explicitely declared in the program. All the other types can only be generated automatically (usually during typechecking).

The types defined below are not metatypes as such, but are used in metatype construction. They are here for completeness of the specification.

```
type T_OpenTypeOrTV subtype of T_Object;

type T_OpenType subtype of T_OpenTypeOrTV {
    primaryFunctor() : T_Type;                  // Primary functor
    children()       : T_List(T_OpenTypeOrTV);  // Children
};

type T_TypeVariable subtype of T_OpenTypeOrTV;

type T_TypeConstraint {
    left()           : T_OpenTypeOrTV;          // left side of the constraint
    right()          : T_OpenTypeOrTV;          // right side of the constraint
};

infinite class C_OpenType implements T_OpenType { ... };
infinite class C_TypeVariable implements T_TypeVariable { ... };
infinite class C_TypeConstraint implements T_TypeConstraint { ... };
```

The following is the type of implementation types:

```
type T_ImplementationType subtype of T_PartiallyOrdered(T_ImplementationType) {
    name()          : T_String;       // The name
    description()   : T_String;       // Documentation
    classes()       : T_Set(T_Class); // All associated classes
    subtypes()      : T_Set(T_ImplementationType);  // Immediate subtypes
    supertypes()    : T_Set(T_ImplementationType);  // Immediate supertypes
    layout()        : T_String;       // host language description
                                      //  of an object layout
};

class C_ImplementationType implements T_ImplementationType { ... };
```

A.4.6 Class types and the metaclasses

```
type T_InfiniteClass(covar X) subtype of T_Object {
    name()              : T_String;       // The name
    type()              : T_Type;         // Associated type
    contains(T_Object)  : T_Boolean;      // Is this an object of this class?
    itype()             : T_ImplementationType;  // Associated implementation type
    extends()           : T_Set(T_InfiniteClass(T_Object));  // Classes that
                                                 // this one extends
};

class C_InfiniteClass implements T_InfiniteClass { ... };

type T_FiniteClass(covar X) subtype of T_InfiniteClass(X) {
    extent()          : T_Set(X);       // Shallow extent
};

class C_FiniteClass implements T_FiniteClass { ... };

type T_ManualClass(covar X) subtype of T_InfiniteClass(X) {
    basicNew()          : X;          // Object creation
};

class C_ManualClass implements T_ManualClass { ... };

type T_Class(covar X) subtype of T_FiniteClass(X), T_ManualClass(X);

class C_Class implements T_Class { ... };
```

Appendix B

Implementation Notes

In this section, design considerations related to various aspects of implementation of a database programming language with the type system described in this dissertation are presented. Most of these design decisions were conceived, discussed, and tried out during implementation of the pilot version of TIGUKAT OODBMS described in [ÖPS+95] and [Pet94].

One of the main goals of the TIGUKAT OODBMS is to provide a uniform object-oriented environment. In this environment, every entity is an object that possesses a type that defines its interface and an implementation type that defines its memory layout. An object is referred to by its unique *object identifier*, or an *OID*. The only elementary actions are behavior applications and (lower-level, invisible for the user) read/write operations on object components (attributes).

Section B.1 discusses in-memory OID structure and the object addressing in general. Section B.2 provides several references on efficient multi-method dispatch techniques. Section B.3 describes the role implementation types play in the system. Finally, Section B.4 briefly discusses the effects of adding persistence to the language.

B.1 Object identifiers

Since the only elementary action explicitly available in the proposed language is behavior application, efficiency of the language is crucially dependent on the efficiency of the dispatch mechanism. Since dynamic types of objects are required for the dispatch to operate, access to the object type information sufficient for dispatch purposes should be made as efficient as possible. The most efficient access to type information can be achieved when it is encoded directly within an OID. Then in order to dispatch on a particular object, this object does not have to be accessed at all; in fact, it may not be in the application memory as long as its OID is available.

Thus the first part of an OID is a *type index*. A type index does not encode the complete object type. Rather, it denotes a pair of which the first component is a primary functor of the object's run-time type and the second component is the object's implementation type. Since only the primary functors make a difference for the dispatch process, the information denoted by the type index is sufficient to perform dispatch. The type index is in fact an index into a global *type table* (hence the name). Each entry in the type table is a pair denoted by the respective type index. In the current implementation, the size of the type index field is 2 bytes which allows for 65535 distinct type indices. Since only concrete types receive type indices, this number is likely to be more than sufficient.

The second part of the OID is the object identifier proper called *contents*. The first two bits of this field identify one of three object varieties. For each variety the rest of the contents field is interpreted differently. In the current implementation, the size of the contents field is 6 bytes.

Short atomic objects are the objects that do not maintain state (are *immutable*) and have small enough size to fit into the contents field entirely. An example of such an object is an integer number. By encoding short atomic objects in this manner their identity is automatically guaranteed. For example, no matter how many times an object **5** appears in the program, all its occurrences will have the same OID and will thus be indistinguishable from one another. An OID of a short atomic object can be perceived as a reference into a constant infinite pool of predefined objects. The user can introduce his own short atomic objects by defining an appropriate implementation type for them.

Long atomic objects are immutable objects, but with size large enough not to fit into the contents field. For such objects, the contents field represents the address on the heap where the object is located. Since long atomic objects are immutable, reference sharing problems do not arise. Immutable sets and strings are examples of system-defined long atomic objects. The user can also introduce his own long atomic objects.

Regular objects are mutable objects. For these objects, the contents field represents an index into the global *object table*. An entry in the object table encodes (along with other information) the address of the object on the heap. The advantage of this indirect approach is the ability to resize and move objects freely (including moving to/from persistent storage) and delete objects in a safe manner. The disadvantage is the impact on efficiency made by an additional indirection during each access. Most if not all user-defined object types will fall into this category.

An implementation type can specify the desired interpretation of the **contents** field of its object identifiers by using specifiers **short atomic**, **long atomic**, and **regular**. This specification is placed in implementation types since it is a low-level property not essential for interface or structure specifications.

In this section, the organization of OIDs was described. The next section outlines efficient dispatch techniques suited for the language in question.

B.2 Multiple dispatch

Since the language under consideration features *multiple dispatch* that requires that dispatch be done on the types of all arguments rather than on the type of the receiver only, some of the well-known efficient dispatch techniques such as vtables do not apply.

Multi-method dispatch techniques can be separated into two broad classes: *cache-based* and *table-based*. Cache-based techniques rely on a sort of a branch prediction mechanism to cache in the most common case, but can take a while to dispatch if a cache miss occurs. The technique used in Cecil language is cache-based [Cha92a], [DCG94]. [DGC95] offers some improvements in non-reflexive case.

Table-based techniques rely on some sort of a table where all possible dispatch variants are stored. The organization and access to this table is the differentiating factor between the techniques from this group. These techniques are described in [CT95], [DAS96], [HSPL98], [HLPS99], and [PHLS99]. Out of these, the technique described in [HLPS99] is the simplest to implement and works very well in the presence of type system changes. Techniques described in [HSPL98] and [PHLS99] are more efficient ($O(k)$ where k is the behavior arity), but an additional work is required to adapt them for a reflexive system.

In the next section, the usage of implementation types in low-level specifications is described.

B.3 Implementation types and functions

An implementation type is a description of the object's memory layout. A special predefined implementation type **IT_TgObject** corresponds to the implementation of a regular TIGUKAT object. The objects of this implementation type have an array of slots that can be accessed by a set of primitive functions. Objects that have a number of specific fields in addition to the array of slots are represented as instances of implementation subtypes of **IT_TgObject**.

Atomic objects that are represented directly by contents of OIDs, are instances of implementation types that are *not* subtypes of **IT_TgObject**. The subtyping relationships between these types is determined by their C++implementation. For example, **IT_Character** and **IT_Natural** are subtypes of **IT_Integer** because in C++ (the host language for our implementation) any character and any unsigned number can be treated as an integer. On the other hand, C++ integers can not be treated as C++ real numbers, and therefore **IT_Integer** is not an implementation subtype of **IT_Real**.

Every implementation type provides a function to retrieve a full object type. This is necessary as the type index only gives information necessary for dispatch purposes, which is in general less then the full type (e.g. a list's OID type index would tell this is an object of type `T_List`(X) for some X, but it will not be able to produce the value of X).

B.4 Adding persistence

Persistence has an effect on both global tables and object representation. In order to make databases machine-independent, the following approach was adopted. The representation of an object in the database (on disk) and in memory are different. The translation is provided by a pair of functions (**pack** and **unpack**) associated with an implementation type. Thus, each implementation type may potentially have its own disk representation. Disk OIDs also differ from in-memory ones. On disk, there are only regular OIDs as both short and long atomic OIDs are converted into in-line tagged data. Another distinction lies in the size of type and object indices. Disk structures have much larger index size[1]; indices are converted at the moment the object is packed (unpacked). Thus, each entry of both object table and type index table contains, in addition to other information, the long disk version of the index (or 0 if that index has been created by the program and does not have any disk equivalent yet). Disk indices are assigned at the time a table entry is packed (usually at transaction commit time).

Since schema objects like types, behaviors etc. can also be made persistent, they have their own type indices and are packed and unpacked by their respective pack/unpack routines. Dependencies between schema objects make it necessary to provide a set of rules that determine which objects are to be made persistent when a schema object becomes persistent. For example, if an object is made persistent, its type and class become persistent as well. These rules are described in detail in [ÖPS+95].

In this section, various implementation and low-level system design issues were briefly discussed. When completed, the full implementation of the TIGUKAT OODBMS and its language will most probably produce much more elaborate texts (theses or otherwise) describing its design.

[1]In the current implementation, the disk version of an OID is 19 bytes long.

Appendix C

Module System

This chapter describes the module system proposed for the TIGUKAT OODBPL. It manages name spaces in a manner that is orthogonal to the type system, thus allowing the type system to be free of name space management issues.

The concept of a *module* as a unit of name scoping and information hiding was first introduced in Modula and its much better known offspring Modula-2 [Wir83]. This concept is extensively used in the modern language design.

Languages that make use of this powerful concept include those of Pascal/Modula family such as Ada [Ada95] (under the name of *packages*), DBPL [SM94], Oberon [RW92], Oberon-2 [MW93], Modula-90 [LML+94], and Modula-3 [Har92].

Languages from ML language family also make use of the modules. Modules in these languages, however, are interface specifications rather than modules as such. It has been shown that ML modules can be used in place of interface types in object-oriented programming [TT94]. This family of languages includes Standard ML [MTH90], Galileo [ACO85], Amber [Car86a], and Haskell [HJW+92].

Other languages that utilize modules include TM [BBdB+93], LOOM [BFP96], Theta [LCD+95], Dylan [App94], Modular Smalltalk [Hol93a], TL [MMS94], BETA [MMPN93] (under the name of *fragments*), Java [AG96], and Pizza [OW97] (the latter two use the name *package*).

In [CL95], the usage of modules in BeCecil is described and their interaction with various object-oriented constructs is investigated.

In Jigsaw [Bra92], modules are used as one of the central language concepts. It is argued that modules be used instead of classes.

Modularity and the use of modules is also advocated in [TNG92].

This chapter is organized as follows. Section C.1 discusses the problems related to the coupling of inheritance and name space management in the current object-oriented languages. Section C.2 illustrates the concepts of module export and import. Inner modules are introduced in Section C.3. The algorithm for name resolution in the presence of inner modules is presented in Section C.4. Finally, Section C.5 briefly discusses issues related to the module persistence.

C.1 Information hiding and sharing

Traditionally, in object-oriented languages *classes* were used as a unit of information sharing and hiding. The best known example of this approach is C++ [Str91] with its use of `public`, `private`, and `protected` keywords. Even though there are no compelling reasons for coupling inheritance with name space management, this approach was also adopted in Java [AG96]. However, due to well-known problems related to this approach, Java has a complementary mechanism for dealing with name space management — namely, that of *packages*. Packages eliminate the need for cumbersome `friend` constructs that are used in C++ to overcome the problems related to the coupling of inheritance and name space management units.

In order to illustrate the problems related to such coupling, consider the following example. Let **T_Bank**, **T_InvestmentCompany**, and **T_InsuranceCompany** be the types that define interface of their respective real-world entities. Consider a financial group that has a bank, an investment company, and an insurance company as its components. Let **FinancialRecords** be a message that can be sent to any bank or company to receive financial records of a given institution. We would like to make **FinancialRecords** available *inside* the financial group, but not *outside* of it. In the absence of modules, there is no way to accomplish this functionality unless a new "wrapper" type **T_FinancialGroup** is defined. However, in this case, all access to the bank and the investment and insurance company has to go through the financial group, which means that all public interfaces of the bank, investment and insurance company has to be duplicated in **T_FinancialGroup**. For example, if banks and companies have a publicly accessible method **WithdrawFunds**, that will necessitate the creation of three methods in **T_FinancialGroup**, namely **WithdrawFundsFromBank**, **WithdrawFundsFromInvestment**, and **WithdrawFundsFromInsurance**. This makes the interface definition unnecessarily complex and any further interface modification complicated and error-prone. In the presence of modules, however, a financial group can be represented as a *module* rather than a *class*. That module may export the bank, investment, and insurance companies together with their public interfaces, but not the message **FinancialRecords**.

In the proposed type system, only the question of message (behavior) hiding is relevant, as all instance variables are hidden and can not be directly accessed by any type. Thus, the following discussion concentrates on the modular mechanisms for behavior hiding.

C.2 Export and import

Every module contains *export* and *import* lists. A module can be understood a membrane that stops everything but the names listed in the import list from getting in, and it also stops all names except for those listed in the export lists from getting out.

Every entry in the export table is organized as follows. It has a qualified name of the symbol to export, its (possibly empty) alias that will be used outside the exporting module, and a flag that indicates whether all names are to be exported. The flag only makes sense if this entry exports a module.

An entry in the import table also consists of the qualified name of the imported symbol, its (possibly empty) alias to be used inside the importing module, and a flag that indicates whether all names are to be imported. The flag only makes sense if this entry imports a module.

Example:

```
module M_Bank {
    import all M_Finance, M_FinancialInstitution;
    import M_FinancialInstitution.T_FinancialInstitution as T_Root;
    export M_Bank.T_Bank, M_Bank.WithdrawFunds, ...;
    export all M_FinancialInstitution;
    ...
    type T_Bank subtype of T_Root ...
    ...
};

module M_Main {
    import all M_Bank;
    ...
    T_Bank bank;
    ...
    bank.WithdrawFunds(...);
    ...
};
```

Here the module M_Bank imports everything that is exported by both M_FinancialInstitution and M_Finance, (locally) renaming T_FinancialInstitution imported from M_FinancialInstitution as T_Root. It exports the type T_Bank, the behavior WithdrawFunds, and all symbols exported by M_FinancialInstitution. The module M_Main imports all symbols exported by M_Bank and can therefore use both T_Bank and WithdrawFunds.

Note that this arrangement allows one to define re-export of imported symbols, possibly renaming them along the way. This makes it possible to define more restrictive versions of existing modules (views). For example, the following defines a "customer view" of a bank:

```
module M_CustomerBank {
    import all M_Bank;
    export M_Bank.T_Bank, M_Bank.WithdrawFunds, ...
};
```

The difference between M_Bank and M_CustomerBank is that the latter only allows the usage of bank's methods, but not those of a financial institution.

C.3 Inner modules

In order to design hierarchical systems, a notion of *inner module* is introduced. An inner module is defined inside another (*parent*) module; thus, the hierarchy of inner modules forms a tree. The import list of the inner module contains a special entry named PARENT. If this entry has its "all" flag turned on, the inner module imports all symbols from its parent module, including ones that are not exported. Thus, an inner module has an access to the "internals" of its parent module.

On the other hand, the parent module has access to all symbols exported by the inner module. It does not have access to the symbols that the inner module does not export.

There is an additional restriction placed on the inner module export list. An inner module can not export imported symbols that are not explicitly exported by its parent module. This is done to disallow bypassing export lists by using inner modules.

Example:

```
module M_Bank {
    import all M_Finance, M_FinancialInstitution;
    import M_FinancialInstitution.T_FinancialInstitution as T_Root;
    export M_Bank.T_Bank, M_Bank.WithdrawFunds, ...;
    export all M_FinancialInstitution;
    ...
    type T_Bank subtype of T_Root ...
    ...
    module M_Vault {
        import all PARENT;
        export T_Vault, Get, Put;
        ...
        type T_Vault ... {
            ...
            Get(T_Money request): T_Money received implementation {
                  ...
                  SecurityStandBy();
                  ...
            }
        };
    };
    ...
    module M_Security {
        import all PARENT;
```

```
            export SecurityStandBy, ...;
            ...
        };
        ...
    };
```

Inside M_Bank only behaviors Get and Put of the type T_Vault can be used. However, inside M_Vault all behaviors defined inside M_Bank can be used, including the behavior SecurityStandBy exported into the name space of M_Bank by the module M_Security.

The next section gives the algorithm for name search in the presence of inner modules.

```
type T_Module {
   Parent()      : T_Module;       // Reference to the parent module.
   NameTable()   : T_NameTable;    // Name table
   ImportTable() : T_ImportTable;  // Import table
   ExportTable() : T_ExportTable;  // Export table
   ...
};

type T_NameTable subtype of T_Set(T_NameTableEntry);
type T_NameTableEntry {
   Name()    : T_String;       // Name being defined
   NameDef() : T_NameDef;      // The definition for use by compiler
   Module()  : T_Module;       // Module reference (non-NULL
                               //  if this is a name of an inner module)
};

type T_ImportTable subtype of T_Set(T_IETableEntry);
type T_ExportTable subtype of T_Set(T_IETableEntry);
type T_IETableEntry {
   QName() : T_String;         // Qualified name being imported (exported)
   Alias() : T_String;         // Its alias (can be NULL)
   All()   : T_Boolean;        // Are all names imported (exported)?
                               // Only makes sense if this entry
                               // refers to a module
};
```

Figure C.1: Module data structures

C.4 Name search

Figures C.1, C.2, C.3, and C.4 define the algorithm and the data structures used for name search in the presence of multiple modules. The entry point of the algorithm is the behavior FindSymbol (Figure C.2). For the purposes of this algorithm, it is assumed that every module has a parent module except for a special "root" module that has no imports or exports. This "root" module is considered to be a parent of all modules defined in the global scope. It is also assumed that import table of every module has an entry marked with a predefined name "PARENT" that refers to the parent of the given module.

The algorithm either produces the variable definition along with the module it is located in or one of two error conditions: UNDEFINED or AMBIGUOUS. There is a built-in ambiguity resolution that assigns lower priority to symbols implicitly imported from inner modules. However, this is the only case when one definition supersedes another. In all other cases, the AMBIGUOUS error occurs.

```
// Function: T_Module::FindSymbol
// Purpose: find a symbol in the given module
// Input:
//     self (the receiver) - module to locate symbol in
//     qname  - qualified name of the symbol to search for
// Output:
//     pair of (module,nameDef) where
//         module   - module where the found name definition is located
//         nameDef  - name definition for use by compiler
// Errors:
//     Can throw one of:
//         AMBIGUOUS - the name is ambiguous
//         UNDEFINED - the name is undefined

T_Module::FindSymbol(T_String qname) : (T_NameDef, T_Module) implementation
{
   T_Set(T_Module,T_String) trace := {};
   T_USet(T_Number, T_NameDef, T_Module) results := {};
   T_NameDef resultND;
   T_Module resultMod;
   T_Number resultPri := MAXNUM;
   T_Boolean resultAmbiguous := FALSE;

   FindHere(qname, 0, TRUE, trace, results);

   if (results.IsEmpty) then { throw UNDEFINED; };

   // Choose the result with the highest priority.
   // If there is more than one, this symbol is ambiguous
   for each e in results do {
       T_Number priority;
       T_NameDef nameDef;
       T_Module module;
       (priority, nameDef, module) := e;
       if (priority < resultPri) then {
           resultAmbiguous := FALSE;
           (resultPri, resultND, resultMod) := (priority, nameDef, module);
       } elseif (priority == resultPri) then {
           resultAmbiguous := TRUE;
       };
   };

   if (resultAmbiguous) then {
      throw AMBIGUOUS;
   } else {
      return (resultND, resultMod);
   };
};
```

Figure C.2: Name search

```
// Function: T_Module::FindHere
// Purpose: find all symbol definitions available in the given module
// Input:
//     self (the receiver) - module to locate symbol in
//     qname  - qualified name of the symbol to search for
//     priority - current search priority (0 is the highest)
//     parentSearch - a boolean flag that indicates whether
//                    the enclosing module should be searched
//     trace  - set of (module,name) pairs already tried
// Input/Output:
//     results - updatable set of results. Each result is a triple
//               that consists of:
//         priority - the priority of the successful search
//         nameDef  - name definition
//         module   - module where the found name definition is located

T_Module::FindHere( T_String qname, T_Number priority, T_Boolean parentSearch,
                    T_Set(T_Module, T_String) trace,
                    T_USet(T_Number, T_NameDef, T_Module) results) implementation
{
    // Attempt to locate the symbol in the name table
    // and inner modules
    for each e in NameTable do {
        if (e.Name == qname) then {
            results.insert((priority,e.NameDef,self));
        } elseif (e.Module != NULL) then {
           if (e.Name.IsPrefixOf(qname)) then {
               T_String nameTail := qname.RemovePrefix(e.Name);
               e.Module.FindExternal(nameTail,priority,trace,results);
           } else {
               e.Module.FindExternal(qname,priority + 1,trace,results);
           };
        };
    };

    // Attempt to locate the symbol among imports
    for each e in ImportTable do {
        if (e.QName == "PARENT" and not parentSearch) then { continue; };
        T_String name := if (e.Alias != NULL) then { e.Alias; } else { e.QName; };
        if (name.IsPrefixOf(qname)) then {
           T_String nameTail := qname.RemovePrefix(name);
           if (e.All or nameTail == "") then {
               T_Set(T_Module,T_String) newTrace := {(self,qname)};
               Parent.FindHere(e.QName + nameTail,priority,TRUE,newTrace,results);
           };
        };
    };

};
```

Figure C.3: Name search in the given module

```
// Function: T_Module::FindExternal
// Purpose: find all symbol definitions exported by the given module
// Input:
//     self (the receiver) - module to locate symbol in
//     qname  - qualified name of the symbol to search for
//     priority - current search priority (0 is the highest)
//     trace  - set of (module,name) pairs already tried
// Input/Output:
//     results - updatable set of results. Each result is a triple
//               that consists of:
//         priority - the priority of the successful search
//         nameDef  - name definition
//         module   - module where the found name definition is located

T_Module::FindExternal( T_String qname, T_Number priority,
                   T_Set(T_Module, T_String) trace,
                   T_USet(T_Number, T_NameDef, T_Module) results) implementation
{
    // Fail if already visited this place
    if (trace.Includes((self,qname))) then {
        return;
    } else {
        trace.Insert((self,qname));
    };

    for each e in ExportTable do {
        T_String name := if (e.Alias != NULL) then { e.Alias; } else { e.QName; };
        if (name.IsPrefixOf(qname)) then {
             T_String nameTail := qname.RemovePrefix(name);
             if (e.All OR nameTail = "") then {
                  FindHere(e.QName + nameTail, priority, FALSE, trace, results);
             };
        };
    };

};
```

Figure C.4: External name search in the given module

Due to the restriction mentioned in the previous section (an inner module can not export symbols from its parent), the behavior **FindHere** has an additional argument **parentSearch** which is set to **FALSE** every time an "external" search is performed. This prevents the algorithm from searching the parent hierarchy of a given module.

The trace is necessary to avoid infinite recursion. The reason why such a situation can occur is the fact that there are no acyclicity restrictions placed on the import module hierarchy. This significantly facilitates the design process. The trace is reset in **FindHere** when the search proceeds to the parent module to reduce the amount of testing required. This optimization is valid since only "internal" searches are allowed to look in the parent module, thus infinite cycles do not occur when the algorithm goes "up" the parent hierarchy.

The data structures depicted in Figure C.1 are related to a single module. The name table contains the set of names defined in the given module, including any inner modules present. The import (export) table represents the set of all imports (exports) of the given module. **Parent** refers to the parent module.

C.5 Adding persistence

Modules, types, behaviors, and all the other schema entities are objects and can therefore be stored in the database and retrieved from it. In order to indicate that a particular module is persistent, the **PERSISTENT** keyword can be added to its definition. If a module is persistent, then all types, behaviors, classes, functions, and inner modules defined in it are also persistent. Only top-level modules can be declared persistent. If a module is persistent, then all modules from which it imports should also be declared persistent. These restrictions ensure that persistent module's dependencies persist along with the module itself.

In this chapter, the module system and the algorithm for name resolution have been described. This module system is orthogonal to the type system and therefore does not affect typechecking. At the same time, the elaborate import/export mechanism along with inner modules provides a powerful way to deal with information hiding and sharing at the module level. The module system is also capable of modeling views which is one of the major schema information management mechanisms in traditional database management systems.