University of Alberta

The Logic of Type Specifications: Typechecking Parametric and Inclusion Polymorphism

by

Yuri Leontiev, M. Tamer Özsu, Duane Szafron

Technical Report TR 98–01
March 1998

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

The Logic of Type Specifications: Typechecking Parametric and Inclusion Polymorphism

Y. Leontiev, M. Tamer Özsu, Duane Szafron
Laboratory for Database Systems Research
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada T6G 2E1
{yuri,ozsu,duane}@cs.ualberta.ca

Abstract

In this paper we present a type system that combines inclusion and parametric polymorphism with behaviors (multi-methods) and precise function typing. Type declarations allow user-definable variance specification of type parameters and user-definable subtyping between types of different kind. Our approach involves use of type specification logic which translates type specifications into types. Types are computable values. Type computation of types generated by the logic results in precise function and behavior typing. As a proof of concept, a toy language with its syntax, semantics, and subject reduction theorem is presented.

1 Introduction

In the past decade the type safety of programming languages has been a focus of prolific research activity. One of the major problems in this area is the development of a type system that would statically ensure type safety and would still be expressive enough to deal with today's advanced modeling and development requirements. It is well known that expressive power and convenience of a type system sometimes, if not in most cases, make static type safety very difficult to achieve. It therefore comes as no surprise that many languages sacrifice static type safety for flexibility and expressiveness of their type systems (e.g Eiffel [Mey88], BETA [MMMP90]).

The recent advancements in the area of type systems make it possible to design an expressive type system that supports static type safety. One of the main ideas that allow us to achieve this goal is the idea of *polymorphism*. Parametric polymorphism is used as a central design principle behind very expressive type systems of ML [MTH90] and many languages influenced by it, such as Napier88 [MBC+96] and Machiavelli [BO96]. Inclusion polymorphism, on the other hand, is used in modern object-oriented languages. Recently, many attempts have been made to combine the expressive power of these two widely used forms of polymorphism in a single type system (PolyTOIL [BSG95], LOOP [ESTZ95], Sather [SOM93], Theta [DGLM95], TooL [GM96], and TL [MMS94]).

However, the full potential of such a powerful combination has yet to be realized. In the presence of two forms of polymorphism, the interaction between them must be identified. This includes the subtyping between parametric types of the same shape, which can be covariant (such as non-updatable sets), contravariant (output streams), or novariant (updatable sets). It also includes subtyping relationships between different parametric types (such as $\texttt{Set}(X) \leq \texttt{UpdatableSet}(X)$) and between parametric and ordinary types (such as $\texttt{String} \leq \texttt{Array}(\texttt{Char})$ or $\texttt{Collection}(\texttt{Collection}(\texttt{X})) \leq \texttt{CollectionOfCollections}$). Each of the languages mentioned above fixes the way the parametric types are treated in terms of variance. PolyTOIL and TooL allow only covariant parametrizations, while Sather and Theta — only novariant ones. Only Theta, Sather, and TooL allow user-specifiable subtyping relationships between different parametric types.

Another important feature of a polymorphic type system is its ability to provide *precise function typing*. This is needed when the return type of a function depends on types of its arguments. The standard technique employs *type variables* that convey type information from argument to result type of a function. This approach is used in ML [MTH90] and many other languages.

The presence of multiple dispatch makes precise function typing significantly more complicated. Multiple dispatch is sometimes defined in terms of *multi-methods*, and the run-time dispatch mechanism is required to pick the "most suitable" method. The problems of static typing of multi-methods have been discussed in [ADL91], [CL94], [CL96], [Ghe91], and others. However, none of these papers discusses the combination of multi-methods and precise function typing in the form of free variables in type specifications.

The ability to specify a wide range of possible interactions between two kinds of polymorphism, as well as precise function typings, greatly enhances the expressive power of the type system. In this paper, we present a mechanism that allows us to specify such interactions, give precise typing to functions and behaviors, and statically type-check the resulting programs.

The mechanism proposed in this paper is designed for object-oriented languages with multiple dispatch (where types of all arguments, not just the receiver, are taken into account). It can be easily adopted for single-dispatched languages as well, but works better in the presence of multiple dispatch. As far as we are aware, the combination of multiple dispatch, parametric polymorphism, and static type checking is novel to our work.

The essence of our approach is the treatment of type specifications as logical formulae (types) in an appropriate variant of type theory. The types of objects are then obtained as proofs of their respective type

specifications. The types generated in this manner can represent functions from types to types. For example, the type specification $X \to X$ has a proof $\lambda x.x$, which is a type of functions that produce the result of the same or lesser type then that of the receiver.

The paper is organized as follows: in Section 2, we describe the type and subtype specifications that are allowed in our framework and introduce the notion of *the user type graph*. Section 3 describes type specifications and exemplifies their intended meaning. Types, subtyping, and the type reduction are described in the Section 4. The Section 5 describes the *type specification logic* that is used to obtain types from type specifications. Later, in Section 6, we introduce a toy language that is used to illustrate our approach, with its syntax and natural semantics. The material in the Section 7 deals with dispatch and behavior consistency issues and includes the subject reduction theorem. Finally, Section 8 concludes the paper and outlines the future research directions.

2 User-defined types

In this section, we will describe the types that are declared by the user. We will also describe the subtype declarations and special, predefined types.

User-defined types (ordinary and parametric) along with a few special predefined types form the basis of the type hierarchy. User-defined types can be *ordinary* or *parametric* and are declared using *type declarations*, while subtyping relationships between them are defined by *subtype declarations*.

In this section we will describe the type and subtype declarations allowed in our framework. We will also define the *user type graph*, which is an auxiliary structure automatically generated from the declarations and used in type-checking and type inferencing. We will require the user type graph to be *consistent* (one of the consistency condition being the graph acyclicity). The formal definition of consistency will be presented in the Section 2.3.

We will start by describing the predefined *special types*, then we will proceed to the declarations and finish by the definition of the user type graph, its consistency, and auxiliary functions defined over it.

2.1 Special predefined types

Several special types are considered to be predefined:

1. The type $\top$. This is the supertype of all other types. $\top$ is the type of run-time type errors[1] ($\mathsf{err}^\top$).
2. The type $\bot$. This is the subtype of all other types. $\bot$ is the type of run-time errors ($\mathsf{err}^\bot$) generated by primitives[2].
3. The type `Object`. This is a supertype of all types in the system except for $\top$. The statement `X` $\leq$ `Object` is interpreted as "`X` is a valid object type".
4. The type `Behavior`. This type is a supertype of all behavior and function types (defined in Section 4).
5. The type `RegObject`. `RegObject` is a supertype of all user-defined types except for `Unit`.
6. The type `Unit`. This type has no supertypes except for `Object` and $\top$ and no subtypes except for $\bot$. The only object of this type is `unit`. This type is used for procedure and command return values.

[1] Thus objects of this type can never occur in a successfully typechecked program.

[2] Potentially, objects of this type can serve as *exceptions* as, once produced, they propagate all the way up to the highest level without causing type errors. This potentially valuable treatment of $\bot$ is outside the scope of this paper.

7. The parametric types $\texttt{Product}_n(X_1, \ldots, X_n)$ $(n > 1)$. These types have no supertypes except for `RegObject` and no subtypes except for $\bot$. The parameters are covariant. Objects of these types have the form $\langle \texttt{o}_1, \ldots, \texttt{o}_n \rangle$ and are used to deal with multiple arguments[3].

The special types and their place in the type hierarchy are depicted in the Figure 1 (except for products that are in the user-defined type hierarchy, but are predefined and therefore special).

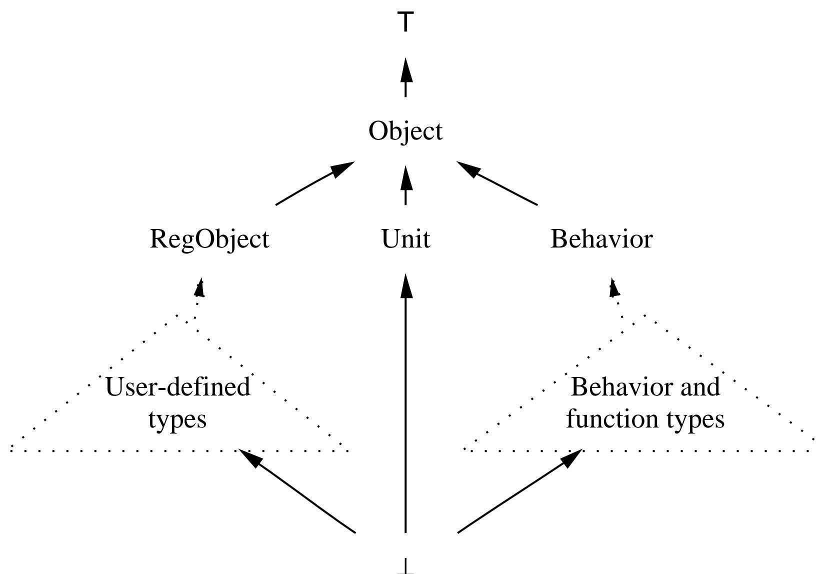

Figure 1: Special types and their place in the type hierarchy

Note that the types $\top$ and $\bot$ do not have "normal" names and thus can not be explicitly used in the program. This is a design decision that can be justified by the following consideration: since both these types are types of errors, they should not occur in a correct program.

Having described the special types, we now proceed to the "normal" user-defined types.

2.2 User type declarations

First, we will establish some terminology and introduce the notation to be used for user-defined types. We will denote user-defined types as `T`. Parametric types will be written as $\texttt{P}(A_1, \ldots, A_n)$, where A_i are type parameters and `P` is called *the head*. The head of an ordinary type is that type itself. A set of all parametric types with the same head will be called *a family of parametric types*.

Both parametric and ordinary types are declared by *type declarations* that have the following syntax:

$\langle$`type-decl`$\rangle$ ::= `type T` | `type P(`$\langle$`tpar-specs`$\rangle$`)`
$\langle$`tpar-specs`$\rangle$ ::= $\langle$`tpar-spec`$\rangle$ | $\langle$`tpar-specs`$\rangle$, $\langle$`tpar-spec`$\rangle$
$\langle$`tpar-spec`$\rangle$ ::= $\langle$`tpar-var-spec`$\rangle$ | $\langle$`tpar-var-spec`$\rangle$ $\leq$ `T`

[3]Note that there is no subtyping between product types of different arities as we do *not* want to be able to call a behavior with four arguments if it was designed to accept three.

⟨`tpar-var-spec`⟩ ::= 0 | + | -

A parametric type declaration is thus the definition of a family of parametric types. Each parameter is characterized by its *variance specification* and its upper bound. An upper bound is always a non-parametric user-defined type.

Variance specifications denote subtyping relationships between parametric types of the same family. Type parameters marked + are covariant, those marked − are contravariant, and those marked 0 are novariant.

For example, `type Dictionary`($+ \leq$ `String`, $+$) is a declaration of a family of (non-updatable) dictionary types with two covariant parameters, the first being required to be a subtype of `String`. We assume that the first argument is the key type and the second one is the value type[4].

The subtyping relationship between types is specified by variance specifications (for within-family relationships) and by explicit *subtype declarations*. Subtype declarations have one of five forms: ordinary-ordinary, parametric-ordinary, ordinary-parametric, parametric-parametric, and parametric-any. We will consider all these forms along with their syntax and semantics in turn. The following is the syntax of a subtype declaration:

⟨`subtype-decl`⟩ ::= ⟨`tt-decl`⟩|⟨`pt-decl`⟩|⟨`tp-decl`⟩|⟨`pp-decl`⟩|⟨`pa-decl`⟩

The first form of subtype declaration establishes an ordinary type as a subtype of another ordinary type. A subtype declaration of this form is written as

⟨`tt-decl`⟩ ::= `T` $\leq_u$ `T`$'$

For example, `Integer` $\leq_u$ `Real`.

The second form of subtype declaration (⟨`pt-decl`⟩) is the definition of a subtyping relationship between a parametric type and its non-parametric supertype.

⟨`pt-decl`⟩ ::= `P`(⟨`pt-par-decls`⟩) $\leq_u$ `T`
⟨`pt-par-decls`⟩ ::= ⟨`pt-par-decl`⟩ | ⟨`pt-par-decls`⟩, ⟨`pt-par-decl`⟩
⟨`pt-par-decl`⟩ ::= `T`$'$ | $*$ | `P`$'$(⟨`pt-par-decls`⟩)

The number of ⟨`pt-par-decls`⟩ must match the definition of the appropriate type. The star $*$ has a meaning of a wildcard that matches anything. It can be considered as a universally quantified unique anonymous variable. This form of subtype declaration has an additional semantic restriction: The ⟨`pt-par-decl`⟩ in the i-th position of the parametric type `P` must be a subtype of $Max_{P,i}$ (the upper bound of the i-th parameter of `P`, as defined by its declaration). For example, the declaration `Dictionary(Text,` $*$`)` $\leq_u$ `TextDictionary` establishes `TextDictionary` as a common supertype of all types of the form `Dictionary(Text,` X`)`, where X is any type, provided that `Text` is a subtype of `String`.

Yet another form of the subtyping declaration, ⟨`tp-decl`⟩, is the declaration of a subtyping relationship between a non-parametric type and its parametric supertype. It has the following syntax:

⟨`tp-decl`⟩ ::= `T` $\leq_u$ `P`(⟨`tp-par-decls`⟩)
⟨`tp-par-decls`⟩ ::= ⟨`tp-par-decl`⟩ | ⟨`tp-par-decls`⟩, ⟨`tp-par-decl`⟩
⟨`tp-par-decl`⟩ ::= `T`$'$ | `P`$'$(⟨`tp-par-decls`⟩)

The absence of wildcards here is the only difference between this form of subtype declaration and the previous one. For example, the declaration `String` $\leq_u$ `List(Character)` makes `String` a subtype of `List(Character)`. Note that since the parametric type here is fully instantiated, it is impossible to make an ordinary type a subtype of an infinite number of parametric types.

[4] Each parametric type declaration `P(...)` implicitly defines constants n_P, $V_{P,i}$, and $Max_{P,i}$. n_P is the arity of P, $V_{P,i}$ is the variance annotation of the i-th parameter of P, and $Max_{P,i}$ is the upper bound for it. If not explicitly given, $Max_{P,i}$ is `Object` for covariant (+) positions and `RegObject` for all the others (− and 0). If explicitly given, the upper bound should be $\leq_c$ `RegObject`.

The reasons why wildcards are disallowed here can be illustrated by the following example. Let us assume that we allow wildcards and have the following definitions[5]:

type UpdatableContainer(0);
type EmptyContainer;
EmptyContainer $\leq_u$ UpdatableContainer($*$);
behavior *B_getOne* {UpdatableContainer(X)$\rightarrow X \ldots$}

Then application of *B_getOne* to an object of type EmptyContainer is type-correct, but what result type is it supposed to produce? Clearly, it can produce neither $\top$ nor $\bot$ since then the behavior *B_getOne* would become non-monotonic (as UpdatableContainer is novariant) and thus prone to run-time type errors. To prevent situations like this, wildcards are disallowed in this form of subtype declaration.

The most complicated form of subtype declaration is ⟨pp-decl⟩ which establishes a subtyping relationship between a parametric type(s) and its parametric supertype(s). The following is its syntax:

⟨pp-decl⟩ ::= P($X_1, \ldots, X_{n_P}$) $\leq_u$ P′(⟨pp-par-decls⟩)
⟨pp-par-decls⟩ ::= ⟨pp-par-decl⟩ | ⟨pp-par-decls⟩, ⟨pp-par-decl⟩
⟨pp-par-decl⟩ ::= X_i | T | P″(⟨tp-par-decls⟩)

X_i are variables (implicitly universally quantified). Each variable must appear once on the left and no more then once on the right. There is also one additional semantic restriction: if the variables on the left and on the right of the subtyping relationship are annotated by $\mapsto^+$ (defined in Section 3, then if a variable X_i is annotated by $+(-)$ on the left, it must also be annotated by $+(-)$ on the right. Note that only "constant" parametric types (⟨tp-par-decls⟩) are allowed inside ⟨pp-par-decl⟩. The following are examples of this form of subtype declaration:

type Set(+);
type Bag(+);
type UpdatableBag(0);
Bag(X) $\leq_u$ Set(X);
UpdatableBag(X) $\leq_u$ Bag(X);

These semantic restrictions are not self-evident, so we will look into them in more detail. The requirement that no new variables appear on the right is equivalent to the absence of wildcards in ⟨tp-decl⟩ and was discussed earlier. In order to justify the requirement dealing with variance annotations, consider the following example. Let us have the following definitions:

type UpdatableContainer(0);
type SpecialContainer(+);
type Person;
type Student;
Student $\leq_u$ Person;
SpecialContainer(X) $\leq_u$ UpdatableContainer(X);
behavior *B_update* {(UpdatableContainer(X), X)$\rightarrow$UpdatableContainer(X) $\ldots$}

Then if we take an object of type SpecialContainer(Student) and apply the behavior *B_update* to it, what result type should we get? At first sight, it should be UpdatableContainer(Student) as

$$\texttt{SpecialContainer(Student)} \leq_u \texttt{UpdatableContainer(Student)}$$

But on the other hand

$$\texttt{SpecialContainer(Student)} \leq_u \texttt{SpecialContainer(Person)} \leq_u \texttt{UpdatableContainer(Person)}$$

[5] Behavior type specifications will be discussed in the Section 3

so the result should also be `UpdatableContainer(Person)`. Since `UpdatableContainer` is novariant, these two results are incompatible. This happened because the variance restriction was violated (X on the left side is covariant (+) while X on the right side is novariant(0))[6].

The last form of subtype declaration is ⟨`pa-decl`⟩. It establishes a subtyping relationship between a parametric type and its argument. The following is the syntax of ⟨`pa-decl`⟩:
⟨`pa-decl`⟩ ::= $\mathtt{P}(X) \ \leq_u X$

Here `P` is a parametric type with a single parameter which must be either covariant (+) or novariant (0). X here is a variable. An example of a subtype declaration of this sort is the declaration of a type of variables:

type `Var`(0);
`Var`(X) $\leq_u$ X;

which allows us to use objects of this type wherever the object of the argument type is expected. This is a natural thing to do, but many type systems require explicit dereferencing operation to achieve this effect. The restrictions on this form of subtype declaration are analogous to those placed on ⟨`pp-decl`⟩ if we consider the stand-alone variable X on the right as a parametric type `Id`(X) with a single covariant parameter[7].

Thus far we have described the type and subtype declarations allowed by our framework along with their respective semantic restrictions. Apart from the conditions related to each of the five forms of subtype declarations, there are also certain restrictions on the complete set of these declarations in the type system. We will specify them in the next section using the notion of *the user type graph*.

2.3 The user type graph and auxiliary functions

The *user type graph* (denoted $\mathcal{G}$) is a graph with nodes labeled by user-defined types (both parametric and ordinary) and directed edges produced (and labeled) by subtype declarations. We will use it not only to verify the correctness of user type specifications, but also during type-checking and other manipulations with types.

Definition 2.1. *The user type graph:* If $\{\mathbf{type}\ \mathtt{T}_k(\ldots)\}$ are type declarations where each type is declared no more than once and $\{\mathtt{T}_i^1(\ldots) \leq_u \mathtt{T}_i^2(\ldots)\}$ are subtype declarations, then the user type graph $\mathcal{G}$ is defined as follows:

1. For each type declaration **type** $\mathtt{T}_k$ there is a vertex in the graph. Thus, every parametric family is represented in $\mathcal{G}$ by a single vertex while every ordinary type has its own corresponding vertex.
2. For each subtype declaration $\mathtt{T}_1(\ldots) \leq_u \mathtt{T}_2(\ldots)$ there is an edge from the vertex $\mathtt{T}_1$ to $\mathtt{T}_2$ in the graph
3. Subtype declarations of the form $\mathtt{Q}(X) \leq_u X$ are *not* reflected in $\mathcal{G}$

□

Note that according to this definition there may be more than two edges between two nodes in the graph. We require that the user type graph is *consistent*. The following is the definition of consistency.

Definition 2.2. *Consistency of the user type graph:* The user type graph $\mathcal{G}$ is consistent iff

[6] The other (implicit) restriction is the fact that the "depth" of parametricity on the right-hand side is restricted to 1. This is done to simplify the definition of functions conv, $\searrow$, and $\nearrow$ (described in the Section 2.3). This restriction can be lifted.

[7] Note that a subtype specification of this form actually produces an infinite number of infinite chains of subtyping:

$$\ldots \leq_u \mathtt{Var(Var(X))} \leq_u \mathtt{Var(X)} \leq_u \mathtt{X}$$

for any type `X`.

1. $\mathcal{G}$ is acyclic as a directed graph
2. If Q participates in the subtype declaration of the form $\texttt{Q}(X) \leq_u X$ ($\langle$pa-spec$\rangle$), then Q is an isolated vertex in $\mathcal{G}$
3. If P is parametric and there are two (directed) paths from a vertex X to P in $\mathcal{G}$, then the two paths share a final segment of non-zero length

□

The first condition protects against "subtyping cycles". The second one makes sure that infinite subtyping chains produced by subtype specifications of the fifth form do not interfere with the rest of the graph. Finally, the third condition ensures that user-specified subtyping declarations do not produce a situation where the way up from a type to a parametric type in $\mathcal{G}$ can produce different results if different paths are taken. This is the same problem the local semantic restrictions placed upon various forms of subtype declarations are designed to protect from.

The user type graph is used to define additional functions conv, $\searrow$, and $\nearrow$. The former one is used for type and subtype computations (Section 4) while the latter two are used for domain intersection and ultimately for behavior consistency checking (Section 7).

The function $\mathrm{conv}(t, P)$ converts a type t to a given supertype. For example, if we have $\texttt{Bag}(X) \leq_u \texttt{Set}(X)$, then $\mathrm{conv}(\texttt{Bag(Person)}, \texttt{Set}) = \texttt{Set(Person)}$. The consistency conditions above ensure that this is indeed a function, i.e. its result is unambiguous. The function $\nearrow$ plays a similar role but returns a set (e.g. $\texttt{Bag(Person)}\nearrow\texttt{Set} = \{\texttt{Set(Person)}\}$). In case of $\nearrow$ the set always consists of only one element. The function $\searrow$ performs a similar function, but goes down the type hierarchy instead of going up. This function can return a set with several elements. In our example, $\texttt{Set(Person)}\searrow\texttt{Bag} = \{\texttt{Bag(Person)}\}$. The complete definitions of these functions are given in the Appendix A[8].

In this section, we have described the type and subtype declarations allowed by our framework along with the restrictions that are placed upon them. We have also introduced the notion of the user type graph and have defined consistency for it. Next, we will describe the *type specifications* used to describe derived types in our framework.

3 Type specifications

Before proceeding to the definition of type specifications, we will first consider a couple of examples. These same examples will be used later to show how the type specification logic and type computations together ensure the precise typing of functions.

[8] For simplicity, we will assume that types Unit, RegObject, and $\texttt{Product}_n$ belong to $\mathcal{G}$ and have the following declarations:

$$\texttt{type Unit;}$$
$$\texttt{type RegObject;}$$
$$\texttt{type Product}_2(+,+);$$
$$\texttt{type Product}_3(+,+,+);$$
$$\vdots$$
$$\texttt{T} \leq_u \texttt{RegObject;}$$
$$\texttt{P}(*,\ldots,*) \leq_u \texttt{RegObject;}$$

for all $\texttt{T} \neq \texttt{Unit}$ and P in $\mathcal{G}$. This way, we will not have to define special subtyping rules for them.

The first example is one of the simplest possible type specifications: $X \rightarrow X$. This is a type specification of a behavior that returns its only argument untouched. Here X is a type variable which is implicitly universally quantified.

A more involved example of a type specification is the type specification of behavior *B_union* that produces a union of two sets. It is written as (`Set`(X),`Set`(Y)) →`Set`(X ∪Y). The type operator ∪ is understood as the least upper bound of two types. The corresponding greatest lower bound operator is ∩. Here both X and Y are type variables and the notation $(T_1, \ldots, T_n)$ will be used throughout the paper as a shortcut for $\texttt{Product}_n(T_1, \ldots, T_n)$. We assume that sets are non-updatable and declared as `type Set`(+).

An interesting example of a type specification is the one of a behavior that adds an element to a set. It is (`Set`(X),X) →`Set`(X). It looks pretty straightforward, but let us consider what happens if the type of the first argument is `Set(Student)` and that of the second argument is `Person` (a supertype of `Student`). The first impression is that this behavior is unapplicable. But there exists an $X =$ `Person` such that the argument types match the ones specified by the behavior. Therefore, the behavior is applicable and the result type is `Set(Person)`. This corresponds to the intuition that by adding just a person to a set of students, we produce a new set that can only be classified as a set of persons. Of course, this would not work with updatable sets, but those are novariant and we would not be able to find an X as in the above example.

The last example of a type specification is that of the behavior *B_apply* that applies its first argument to the second one and returns the result. The specification is $(X \rightarrow Y, X) \rightarrow Y$. Clearly, we can say what the result type is if we know the types of both arguments. For instance, if the first argument is the behavior that adds an element to a set (described above) and the second argument is a pair (product) of an object of type `Set(Student)` and an object of type `Person`, then the result of this behavior will be `Set(Person)` as it is exactly what we'll get according to the previous example.

Now we proceed to the definition of a type specification.

Definition 3.1. *Type specification:* The type specification ($\langle$`TS`$\rangle$) is defined syntactically as follows:

$TS ::= X$ (X is a type variable)
 | T (T is a user-defined type)
 | `Object`
 | `Behavior`
 | $P(TS_1, \ldots, TS_{n_P})$ (P is a user-defined parametric type with arity n_P)
 | $TS_1 \cup TS_2$
 | $TS_1 \cap TS_2$
 | $TS_1 \rightarrow TS_2$
 | (TS)
 | $(TS_1, \ldots, TS_n)$ (shortcut for $\texttt{Product}_n(TS_1, \ldots, TS_n)$)

□

The operators ∩ and ∪ are, respectively, the greatest lower bound and the least upper bound operators. The operator → is the function operator. The variables in a type specification are implicitly universally quantified.

The type specifications as defined here are meant to be written by the user in the program. There is also a notion of *annotated type specification* which is only used by the type-checking mechanism described later. The annotated type specification differs from the normal one in that all variables are annotated with their variance specifications $(0, +,$ or $-)$. The transformation $\mapsto^{+}$ (defined in the Appendix B) transforms a correct type specification into an annotated type specification.

The transformation is straightforward except for one thing: arrow types are only allowed in covariant positions. This restriction is necessary for the definition of computable subtyping given in Section 4.2. It is also intuitively justified as a behavior type in novariant position "fixes" the behavior type too precisely to be useful. The same is true for contravariant occurrences.

In this section we have introduced the type specifications and gave several examples to illustrate their meaning. The precise semantics of the type specifications will be established in Section 5 in terms of types that will be described next.

4 Types and subtyping

In this section, we will describe types that will be used to give the semantics of type specifications in Section 5. We will also describe two subtyping relationships (one is used for computation, the other - for theorems and theoretical considerations).

4.1 Types

In our framework, types are computable. The computation on types gives precise typings to function and behavior applications. For example, if a behavior has type specification $X \rightarrow X$, the type of such a behavior will be $\lambda x.x$. Being applied to any argument type, it will produce the result type. Only monotonic (w.r.t subtyping[9]), double-strict (w.r.t $\top$ and $\bot$), total functions are considered. In order to see how we can require that all functions be total, consider the following example. Let us assume that a behavior *B_plus* has the type specification `Integer` $\rightarrow$`Integer`. Then that behavior has a type $\lambda x.(\mathrm{cond}(x \leq \texttt{Integer}); \texttt{Integer})$ which according to the type computation rules to be discussed below is reduced to `Integer` if applied to a type that is a subtype of `Integer` and is reduced to $\top$ otherwise. It is easy to see that extending types to total functions in this manner preserves both the usual notion of function subtyping (domain contravariance) and monotonicity.

The following is the definition of a type.

Definition 4.1. *Type:* a is a type iff
$a ::=$ x (x is a variable)
| t (T is a user-defined type)
| `Object`
| `Behavior`
| $\top$
| $\bot$
| $p(a_1, \ldots, a_{n_P})$ (P is a user-defined parametric type, n_P is its arity)
| $a_1 \cup a_2$
| $a_1 \cap a_2$
| $\lambda x.a$ (only functions monotonic w.r.t $\leq_f$ are allowed)
| $a \rightarrow b$ (shortcut for $\lambda x.(cond(\{x \leq a, a \leq \texttt{Object}\}); b))$
| $\mathrm{rev}(p, i, a)$ (P is a user-defined parametric type, i is a number from 1 to n_P)
| $\mathrm{conv}(p, a)$ (P is a user-defined parametric type)
| $\mathrm{apply}(a_1, a_2)$

[9] More precisely, $\leq_f$ which will be described later in this section.

$\mid \text{cond}(C); a$ (C is a set of conditions of the form $a_1 \leq a_2$ or $a_1 = a_2$)

□

Here rev, conv, apply, and cond are auxiliary functions. The meaning of $\text{conv}(p, a)$ has already been discussed in Section 2.3. $\text{rev}(p, i, a)$ is similar except that it extracts the i-th argument from the resulting parametric type. The function $\text{cond}(C); a$ checks the conditions in C (using $\leq_c$) and returns $\top$ if they are not satisfied and the result of a otherwise. The function $\text{apply}(f, a)$ applies lambda-abstraction which is its first argument to its second argument and returns the result. The precise definitions of these functions along with type computation (reduction) rules are given in Appendix C. The type reduction is denoted $\Downarrow$.

There are several flavors of types, all slightly different from each other. The *extended types* allow the wildcard ($*$) in any position a variable is allowed. The *closed types* are types with no free variables. *Reduced types* are types with all reductions carried out; they do not include rev, conv, apply, or cond. *Argument types* are types obtained by a simple-minded translation of arguments of type specifications (called $\vdash_{TR}$ and defined in the Appendix D). Except for being reduced, they do not contain occurrences of $\top$, $\bot$, and $\cup$ and restrict lambda abstractions to those produced by the arrow operator. Finally, *concrete types* are types that an object can have. These types disallow $\cap$ except for lambda-abstractions, but allow arbitrary lambda-abstractions. All these varieties are defined in Appendix E.

Having described types, we now proceed to the description of subtyping relationships that we define over them.

4.2 Subtyping

We define two subtyping: $\leq_c$ and $\leq_f$. The first one can be computed and is therefore called *computable subtyping*, while the second one has some nice theoretical properties and is called *full subtyping*.

The computable subtyping is used in type computations, because its rules can be used as an algorithm. However, the computable subtyping is restricted in that it can only give a positive result when lambda abstractions on the right are limited to arrow types. Full subtyping is free from this drawback; however, it is not computable. This is the reason why we have two subtypings, one (computable) stronger than the other (full).

The rules for computable subtyping are presented in Figure 2.

Subtyping of parametric types (rule $p^>$) and ordinary user-defined types (rule $Const^>$) makes use of auxiliary function conv defined over the user type graph (Section 2.3). The rule $Const^>$ uses this function directly and the rule $p^>$ indirectly, via the function rev.

The rule $Var^{<}_{\texttt{Object}}$ is designed for checking types with unbound free variables. It states that a free variable is assumed to be type-correct. This (and equality between two identical variables by the rule Axiom_α) is the only rule dealing with free variables.

The rules $*^>$ and $*^<$ deal with extended types, i.e. types with wildcards in them. The meaning of these rules is "wildcard matches anything". Since we have no structural transitivity rule, the presence of the wildcard rules does not introduce subtyping cycles for non-extended types.

As can be seen from the rules, the computable subtyping is quite weak in terms of subtyping of lambda abstractions (rule $\lambda^{<>}$) and can only deal with arrow types (rather than with arbitrary abstractions) on the right of $\leq_c$. This weakness does not manifest itself during type-checking (since we disallow arrow types in contravariant and novariant positions), but it complicates the theory.

In order to avoid these complications, we define *full subtyping* as an extension of $\leq_c$ where arbitrary lambda-abstractions can be compared. We then establish relationship between two subtypings as well as

$$\frac{a \leq_c c}{a \cap b \leq_c c} \cap_l^< \qquad \frac{b \leq_c c}{a \cap b \leq_c c} \cap_r^< \qquad \frac{c \leq_c a \qquad c \leq_c b}{c \leq_c a \cap b} \cap^>$$

$$\frac{a \leq_c c \qquad b \leq_c c}{a \cup b \leq_c c} \cup^< \qquad \frac{c \leq_c a}{c \leq_c a \cup b} \cup_l^> \qquad \frac{c \leq_c b}{c \leq_c a \cup b} \cup_r^>$$

$$\frac{[e/y](\mathrm{cond}(C); f_2)\Downarrow \leq_c f_1}{\lambda y.(\mathrm{cond}(C); f_2) \leq_c e \rightarrow f_1} \lambda^{<>}$$
where x is not free in e and f_1

$$\frac{[\bot/x](\mathrm{cond}(C); f)\Downarrow \leq_c \texttt{Object}}{\lambda x.(\mathrm{cond}(C); f) \leq_c \texttt{Behavior}} \lambda^<_{\texttt{Behavior}}$$

$$\frac{\mathrm{rev}(p, 1, c)\Downarrow \ \leq_c^{V_{P,1}} \ a_1 \qquad \ldots \qquad \mathrm{rev}(p, n_P, c)\Downarrow \ \leq_c^{V_{P,n_P}} \ a_{n_P}}{c \leq_c p(a_1, \ldots, a_{n_P})} p^>$$
where $\leq_c^+$ is $\leq_c$, $\leq_c^-$ is $\geq_c$, and $\leq_c^0$ is $=_c$

$$\frac{a_1 \leq_c Max_{P,1} \qquad \ldots \qquad a_{n_P} \leq_c Max_{P,n_P}}{p(a_1, \ldots, a_{n_P}) \leq_c \texttt{Object}} p^<_{\texttt{Object}}$$

$$\frac{\mathrm{conv}(t, c)\Downarrow \leq_c \texttt{Object}}{c \leq_c t} Const^> \qquad \frac{}{t \leq_c \texttt{Object}} Const^<_{\texttt{Object}}$$

$$\frac{}{x \leq_c \texttt{Object}} Var^<_{\texttt{Object}}$$

$$\frac{a \leq_c \texttt{Behavior}}{a \leq_c \texttt{Object}} \texttt{Behavior}$$

$$\frac{}{a \leq_c \top} \top \qquad \frac{}{\bot \leq_c a} \bot$$

$$\frac{}{* \leq_c a} *^< \qquad \frac{}{a \leq_c *} *^>$$

$$\frac{a\Downarrow \text{ is } b\Downarrow \text{ up to } \alpha\text{-conversion}}{a \leq_c b} \mathsf{Axiom}_\alpha$$

$$\frac{a\Downarrow \leq_c b\Downarrow \qquad b\Downarrow \leq_c a\Downarrow}{a =_c b} \leq_c \rightarrow =_c$$

Figure 2: Computable subtyping ($\leq_c$)

several properties of the second one.

Full subtyping ($\leq_f$) is defined by making the following changes to the rules for $\leq_c$:

1. The rules $\lambda^{<>}$ and $\lambda^{<}_{\texttt{Object}}$ are different
2. The rule $\lambda^{=}$ is introduced
3. In all rules, $\Downarrow$ is changed to $\Downarrow_f$

The type computation $\Downarrow_f$ differs from $\Downarrow$ in that all subtype checking occurs via $\leq_f$ rather than via $\leq_c$. The following are the λ rules for $\leq_f$:

$$\frac{\forall e: \quad [e/x]f_1\Downarrow_f \leq_f [e/y]f_2\Downarrow_f}{\lambda x.f_1 \leq_f \lambda y.f_2} \lambda^{<>} \qquad \frac{\exists a: \quad [a/x]f\Downarrow_f \leq_f \texttt{Object}}{\lambda x.f \leq_f \texttt{Behavior}} \lambda^{<}_{\texttt{Behavior}}$$

$$\frac{\forall e: \quad [e/x]f_1\Downarrow_f =_f [e/y]f_2\Downarrow_f}{\lambda x.f_1 =_f \lambda y.f_2} \lambda^{=}$$

Note that while the rules in Figure 2 can be considered as computation rules if read from bottom to top, the addition of the λ rules above destroys this property.

The following theorem establishes the relationship between computable ($\leq_c$) and full ($\leq_f$) subtyping as well as ordering properties of the latter.

Theorem 4.1. *Subtyping:* If t, t' are reduced types, then

1. $t \leq_c t' \Rightarrow t \leq_f t'$
2. $\leq_f$ is a partial order on reduced types

□

Sketch of the proof: The first statement follows directly from comparative analysis of rules for $\leq_f$ and $\leq_c$. The second statement can be proven by structural induction over the derivation of $\leq_f$.

In this section, we have defined computable and full subtyping. We have also shown the latter being the weaker form of the two subtypings and established its ordering properties. We will proceed by defining *the logic of type specifications* — a formal mechanism that allows us to translate type specifications into types.

5 The logic of type specifications

In this section we will describe our approach to automatic generation of types from type specifications. We want that translation to give the type specifications the intuitive meaning described earlier in the Section 3. At the end of this section we will present the theorem that confirms that our translation does have this property. We will also come back to the examples of Section 3 and show how the logic handles them.

The approach we take treats the type specifications as *types* in an appropriate variant of Martin-Löf's type theory, while treating our types as elements of these *types*. Doing so enables us to reduce the task of automatic generation of the types to the task of constructing proofs in an appropriate variant of intuitionistic logic.

The logic of type specifications is presented in Figure 3. In this variant of intuitionistic logic, function types ($\rightarrow$) correspond to implication, union types ($\cup$) correspond to disjunction, and both intersection types

($\cap$) and parametric types ($P(A_1, \dots, A_n)$) correspond to conjunctions. Type constants correspond to truth values.

The type specification logic is defined over annotated type specifications. The proof object produced by the logic for an annotated type specification is a closed type.

Sequents in the logic have the form $\Gamma \vdash a :: A/C$ where Γ is a set of proof-annotated formulae, $a :: A$ is a proof-annotated formula (a is a proof of A) and C is a set of conditions. The conditions are used for argument checking and are generated in the leafs of the proof search and then pushed up to the corresponding λ-abstraction (rule $\rightarrow I$).

Since we have no elimination rule for disjunction ($\cup$), we can use a simplified version of implication-elimination rule ($\rightarrow E$). This makes our logic non-contracting in the sense of Dyckhoff [Dyc92]. Therefore a goal-directed proof search is terminating.

The part of the logic that deals with variance annotations consists of the rules for variables (weakening and axioms). These rules place certain restrictions on the assumptions that can be discarded and produce appropriate conditions in C. For example, the sequent $a :: X^+, b :: X^0 \vdash c :: X^+/C$ can be proven if we put $c = b$, $C = \{a \leq b\}$, but the sequent $a :: X^+, b :: X^- \vdash c :: X^+/C$ can not be proven. In the first case we know that b should be matched exactly by the proof c of X (since it is novariant), and a should be matched by the proof c of X in such a way that $a \leq c$ (as it is covariant). Thus, we should put $c = b$ and require that $a \leq c$ which is exactly what the logic produces. In the second case we need to have c such that $c \leq b$ and $a \leq c$, so even if we know that $a \leq b$, we still have insufficient information to determine what exactly c should be.

The "subexpression" premise of the rule Weakening Var deals with the case of recursive variable dependencies. Consider the type specification $(X, X \rightarrow X) \rightarrow X$. It will be annotated to produce $(X^+, X^+ \rightarrow X^+) \rightarrow X^+$. When we try to prove it we will reach the sequent

$$\mathrm{rev}(\mathtt{Product}_2, 1, x) :: X^+, \mathrm{apply}(\mathrm{rev}(\mathtt{Product}_2, 2, x), \mathrm{rev}(\mathtt{Product}_2, 1, x)) :: X^+ \vdash q :: X^+/C$$

and since the proof object of the first X^+ in the assumption list is a subexpression of the proof object of the second one and there is no proof for X^0 in the assumption list, the weakening rule is unapplicable and the sequent can not be proven. The reason for that is that in a perfect world the proof q should look like an upper bound of the set

$$\{\mathrm{rev}(\mathtt{Product}_2, 1, x), \mathrm{apply}(\mathrm{rev}(\mathtt{Product}_2, 2, x), \mathrm{rev}(\mathtt{Product}_2, 1, x)), \\ \mathrm{apply}(\mathrm{rev}(\mathtt{Product}_2, 2, x), \mathrm{apply}(\mathrm{rev}(\mathtt{Product}_2, 2, x), \mathrm{rev}(\mathtt{Product}_2, 1, x))), \dots\}$$

but we can not express this as a type. Note that the type specification $(\mathtt{UpdatableSet}(X), X \rightarrow X) \rightarrow X$ can be proven (because of novariance of UpdatableSet) and will (after simplifications) yield the proof object

$$\lambda x.(\mathrm{cond}(\mathrm{apply}(\mathrm{rev}(\mathtt{Product}_2, 2, x), \mathrm{rev}(\mathtt{Product}_2, 1, \mathrm{rev}(\mathtt{UpadatableSet}, 1, x))) \\ \leq \mathrm{rev}(\mathtt{Product}_2, 1, \mathrm{rev}(\mathtt{UpadatableSet}, 1, x))); \\ \mathrm{rev}(\mathtt{Product}_2, 1, \mathrm{rev}(\mathtt{UpadatableSet}, 1, x)))$$

Having discussed the properties of the logic, we now present the theorem that establishes the precise meaning of proofs produced by the type specification logic. The corollary shows that function types produced in this manner are indeed types, i.e. they are monotonic w.r.t full subtyping ($\leq_f$).

Theorem 5.1. *Properties of derived types:* If $F, \{A_i\}_i$ are annotated type specifications, $f, \{a_i\}_i$ are types, C is a set of conditions, $\vdash_{TR} f' :: F, \{\vdash_{TR} a'_i :: A_i\}_i$, $\vec{x}$ are free variables in $f', \{a'_i\}_i$, and $\{a_i :: A_i\}_i \vdash f :: F/C$ then

$$f = \min^{\leq_f}_{\vec{u} \in U} \{(\mathrm{cond}(C); [\vec{u}/\vec{x}]_c f')\}$$

$$\frac{\Gamma \vdash x :: A/C_1 \qquad \Gamma \vdash y :: B/C_1}{\Gamma \vdash x \cup y :: A \cup B/C_1 \cup C_2} \cup I$$

$$\frac{\Gamma \vdash x :: A/C_1 \qquad \Gamma \vdash y :: B/C_1}{\Gamma \vdash x \cap y :: A \cap B/C_1 \cup C_2} \cap I \qquad\qquad \frac{\Gamma, x :: A, x :: B \vdash f :: F/C}{\Gamma, x :: A \cap B \vdash f :: F/C} \cap E$$

$$\frac{\Gamma, x :: A \vdash y :: B/C}{\Gamma \vdash \lambda x.(\mathrm{cond}(C_1); y) :: A \rightarrow B/C_2} \rightarrow I$$

$C = C_1 \cup C_2$ where conditions from C_1 contain x

$$\frac{\Gamma \vdash y :: A/C_1 \qquad \Gamma, \mathrm{apply}(x, y) :: B \vdash f :: F/C_2}{\Gamma, x :: A \rightarrow B \vdash f :: F/C_1 \cup C_2} \rightarrow E$$

$$\frac{\Gamma \vdash x_1 :: A_1/C_1 \qquad \ldots \qquad \Gamma \vdash x_{n_P} :: A_{n_P}/C_{n_P}}{\Gamma \vdash p(x_1, \ldots, x_{n_P}) :: P(A_1, \ldots, A_{n_P})/\cup_{i=1}^{n_P}(C_i \cup \{x_i \leq Max_{P,i}\})} PI$$

$$\frac{\Gamma, \mathrm{rev}(p, 1, x) :: A_1, \ldots, \mathrm{rev}(p, n_P, x) :: A_{n_P} \vdash f :: F/C}{\Gamma, x :: P(A_1, \ldots, A_{n_P}) \vdash f :: F/C} PE$$

$$\frac{\Gamma \vdash /C}{\Gamma \vdash t :: T/C} \mathsf{Const}I \qquad\qquad \frac{\Gamma \vdash f :: F/C}{\Gamma, x :: T \vdash f :: F/C \cup \{x \leq t\}} \mathsf{Const}E$$

$$\frac{\{\{x_{i,j} :: X_j^v\}_{i=1,\ldots,i_j} \vdash /C_j\}_{j=1,\ldots,n}}{\{x_{i,j} :: X_j^v\}_{\substack{i=1,\ldots,i_j \\ j=1,\ldots,n}} \vdash /\cup_{j=1}^{n} C_j} \text{Weakening Vars}$$

$$\frac{\{\{x_{i,j} :: X_j^v\}_{i=1,\ldots,i_j} \vdash /C_j\}_{\substack{j=1,\ldots,n \\ j \neq J}} \qquad \{x_{i,J} :: X_J^v\}_{i=1,\ldots,i_J} \vdash y :: X_J^V/C_J}{\{x_{i,j} :: X_j^v\}_{\substack{i=1,\ldots,i_j \\ j=1,\ldots,n}} \vdash y :: X_J^V/\cup_{j=1}^{n} C_j} \text{Axiom Vars}$$

$$\frac{\begin{array}{c}(i_0 \geq 1 \vee i_- = 0 \vee i_+ = 0) \wedge \\ ((x_{v_1,i} \text{ is a subexpression of } x_{v_2,j}) \Rightarrow (i = j \vee v_1 = 0 \wedge v_2 \in \{0, +\}))\end{array}}{\begin{array}{c}\{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash /\cup_{\substack{i,j=1,\ldots,i_0 \\ i \neq j}}\{x_{0,i} = x_{0,j}\} \cup_{\substack{i=1,\ldots,i_0 \\ j=1,\ldots,i_+}}\{x_{+,j} \leq x_{0,i}\} \cup_{\substack{i=1,\ldots,i_0 \\ j=1,\ldots,i_-}}\{x_{0,i} \leq x_{-,j}\} \\ \cup_{i=1,\ldots,i_0}\{x_{0,i} \leq \mathtt{Object}\} \cup_{i=1,\ldots,i_+}\{x_{+,i} \leq \mathtt{Object}\} \cup_{i=1,\ldots,i_-}\{x_{-,i} \leq \mathtt{Object}\}\end{array}} \text{Weakening Var}$$

$$\frac{i_0 \geq 1 \qquad \{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash /C}{\{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash x_{0,1} :: X^V/C} \text{Axiom Var0}$$

$$\frac{i_+ \geq 1 \wedge i_0 = i_- = 0 \qquad \{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash /C}{\{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash x_{+,1} \cup \ldots \cup x_{+,i_+} :: X^+/C} \text{Axiom Var+}$$

$$\frac{i_- \geq 1 \wedge i_0 = i_- = 0 \qquad \{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash /C}{\{x_{v,i} :: X^v\}_{\substack{v=+,-,0 \\ i=1,\ldots,i_v}} \vdash x_{-,1} \cap \ldots \cap x_{-,i_-} :: X^-/C} \text{Axiom Var}-$$

Figure 3: Logic of type specifications

where $U = \{\vec{x} \mid \{a_i \leq_f [\vec{u}/\vec{x}]_c a'_i\}_i\}$. The notation $[a/b]_c c$ denotes *substitution with capture* where free variables of a might become bound in $[a/b]_c c$. For example, $[x/t]_c(\lambda x.t) = (\lambda x.x)$. □

Sketch of the proof: The proof is by showing that the conclusion of every logical rule satisfies the theorem if the premises of that rule do. This is tedious but straightforward. The most tedious case is the rule PE as it requires careful analysis of the function rev.

Corollary 5.1. *Properties of derived arrow types:* If

$$\langle \mathtt{TS} \rangle \mapsto^+ A \rightarrow R$$
$$\vdash f :: A \rightarrow R/C$$
$$\vdash_{TR} a :: A, \qquad \vdash_{TR} r :: R$$
$$C = \{c_i \leq^{v_i} d_i\}_i$$
$$c_1 \leq_c^{v_1} d_1, \ldots, c_n \leq_c^{v_n} d_n$$

then

1. $\mathrm{apply}(f,t)\Downarrow_f = \min_{\vec{u}}^{\leq_f} \{\mathrm{cond}(\{t \leq [\vec{u}/\vec{x}]_c a, [\vec{u}/\vec{x}]_c a \leq \mathtt{Object}\}); [\vec{u}/\vec{x}]_c r)\}$
2. f is monotonic w.r.t. $\leq_f$
3. f is monotonic w.r.t. $\leq_c$

□

Sketch of the proof: The first two statements are straightforward. In order to prove the third one, we notice that the only significant difference between $\leq_f$ and $\leq_c$ is that the subtyping rule $\lambda^{<>}$ ($\leq_c$) works well in one direction only. This direction will not be reversed during subtype checking and type computation because the transformation $\mapsto^+$ only allows arrow types in covariant positions.

Now we will consider the same examples we did in Section 3 in order to see the types produced by the logic. In order to make the generated types easier to understand, we will introduce the following notation: $a{\downarrow}i \equiv \mathrm{rev}(\mathtt{Product}_k, i, a)$ (extraction of the i-th component of a product type, e.g. $(a,b){\downarrow}2\Downarrow = b$) and $\mathtt{P}^{-1}(a) \equiv \mathrm{rev}(\mathtt{P}, 1, a)$ (extraction of an argument of a parametric type, e.g. $\mathtt{UpdatableSet}^{-1}(\mathtt{UpdatableSet}(a))\Downarrow = a$). We will also simplify resulting types to make them more visual.

First we consider the type specification $X \rightarrow X$. Its annotated version is $X^+ \rightarrow X^+$ and the type produced by the logic is $\lambda x.(\mathrm{cond}(x \leq \mathtt{Object}); x)$ which is equivalent to $\lambda x.x$ (we always assume that the argument is a valid type). The Corollary states that we must have $\mathrm{apply}(\lambda x.x, t)\Downarrow_f = \min_u^{\leq_f} \{\mathrm{cond}(\{t \leq [u/x]_c x, [u/x]_c x \leq \mathtt{Object}\}); [u/x]_c x)\}$ It is easy to see that the minimum is achieved when we put $u = t$ (u should be greater than t in order for the condition to be satisfied; out of those $u \geq t$, the minimum u provides the minimum result). Then the minimum is t which is equal to $\mathrm{apply}(\lambda x.x, t)\Downarrow_f$.

The second example is the type specification $(\mathtt{Set}(\mathrm{X}), \mathtt{Set}(\mathrm{Y})) \rightarrow \mathtt{Set}(\mathrm{X} \cup \mathrm{Y})$. The logic produces the proof object $f_1 = \lambda x.\mathtt{Set}(\mathtt{Set}^{-1}(x{\downarrow}1) \cup \mathtt{Set}^{-1}(x{\downarrow}2))$ (after simplifications). This can be seen as an instruction to obtain the result type as a set type of the greatest lower bound of parameters of the first and second arguments of the behavior. For instance,

$\mathrm{apply}(f_1, (\mathtt{Set}(\mathtt{Student}), \mathtt{Set}(\mathtt{Person})))\Downarrow =$

$\quad \mathtt{Set}(\mathtt{Set}^{-1}(\mathtt{Set}(\mathtt{Student})) \cup \mathtt{Set}^{-1}(\mathtt{Set}(\mathtt{Person})))\Downarrow =$

$\quad \mathtt{Set}(\mathtt{Student} \cup \mathtt{Person})\Downarrow =$

$\mathtt{Set(Person)}$
which is exactly the behavior we expect from the union on non-updatable sets.

The type specification ($\mathtt{Set}$(X),X) $\to\mathtt{Set}$(X) produces the proof object $f_2 = \lambda x.\mathtt{Set}(\mathtt{Set}^{-1}(x{\downarrow}1) \cup (x{\downarrow}2))$ It is easy to see that

$$\text{apply}(f_2, (\mathtt{Set(Student)}, \mathtt{Person})){\Downarrow} = \mathtt{Set(Person)}$$

which is the expected result for non-destructive addition of an element to a non-updatable set.

Note that if we had a novariant type $\mathtt{UpdatableSet}$(X) and the type specification ($\mathtt{UpdatableSet}$(X),X) $\to\mathtt{UpdatableSet}$(X) the generated proof object would be

$$f'_2 = \lambda x.(\text{cond}(x{\downarrow}2 \le \mathtt{Set}(\mathtt{Set}^{-1}(x{\downarrow}1))); \mathtt{Set}(\mathtt{Set}^{-1}(x{\downarrow}1)))$$

and an attempt to add a person to an updatable set of students will be tagged as a type error,

$$\text{apply}(f'_2, (\mathtt{Set(Student)}, \mathtt{Person})){\Downarrow} = \top$$

because the condition under cond is not satisfied.

The behavior-application behavior signature $(X{\to}Y, X){\to}Y$ will produce the proof object

$$f_3 = \lambda x.\,\text{apply}(x{\downarrow}1, x{\downarrow}2)$$

Thus if the behavior *B_NonDestructiveSetAdd* has the type generated from the type specification ($\mathtt{Set}$(X),X) $\to\mathtt{Set}$(X) (i.e f_2), then

$\text{apply}(f_3, (\mathit{B_NonDestructiveSetAdd}, (\mathtt{Set(Student)}, \mathtt{Person}))){\Downarrow} =$
$\text{apply}(f_2, (\mathtt{Set(Student)}, \mathtt{Person})){\Downarrow} =$
$\mathtt{Set(Person)}$

The last example is of more general nature. Let us consider two type specifications: $\mathtt{A}{\to}\mathtt{B}$ and $\mathtt{A}'{\to}\mathtt{B}'$, where all the types are variable-free. Then the proof of the first specification will be $f = \lambda x.(\text{cond}(x \le \mathtt{A}), \mathtt{B})$ and that of the second $f' = \lambda x.(\text{cond}(x \le \mathtt{A}'), \mathtt{B}')$. Using the subtyping rules (for $\le_c$) it is easy to verify that $f \le_c f' \iff (\mathtt{A}' \le_c \mathtt{A} \wedge \mathtt{B} \le_c \mathtt{B}')$ which is the standard subtyping rule for function types.

The above examples illustrate the way type specification logic and type computations work together to ensure precise typing of functions.

In this section, we have introduced and discussed the type specification logic that produces computable types from type specifications. We have also presented the theorem that establishes properties of this transformation. Finally, we have given several examples that illustrate how type specification logic and type computation process work together to produce precise typing of functions in our framework.

In the next section, we will consider application of the theory developed so far to type-checking of a toy language.

6 The toy language: syntax and semantics

In this section, we introduce a toy language that we will use to illustrate our approach to type specification and type-checking. We will first describe the syntax of the language and explain the meaning of the language constructs that have not been discussed earlier. Then we will describe the typing rules for the toy language and its natural semantics. This will allow us to introduce notions of *behavior consistency* and *dispatch* and prove the subject-reduction theorem in the next section.

The syntax of the toy language is shown in the Figure 4. The program in the language consists of a set of user type and subtype declarations, a set of behavior definitions, and an expression. We have already discussed user type and subtype declarations (along with their syntax) in the Section 2.

A behavior declaration is a set of associations, where each association consists of a type specification (discussed in the Section 3) and a function. When a behavior is applied to an object, the run-time type of the object is used to choose the appropriate association from the set of associations of that behavior. This process is called *dispatch* and will be discussed in the Section 7. After choosing an association, the function from that association is invoked on the receiver object.

The function in an association can be either a function written in the toy language (⟨**function**⟩) or a primitive function that is referred to by its name. Primitive functions have their own reductions and can work with the *store*. The definition of primitive functions and the store depends on the goals of a language designer. For the theory presented here it is sufficient that primitive function associations and the initial store be *consistent* in the sence described in the Section 7.

For example, let us assume that we have `ColorVector` which is a subtype of `Vector` and the behavior *B_Plus* that is defined as

behavior *B_Plus* {
    (Vector,Vector)→Vector : ⟨**function**⟩$_1$,
    (ColorVector,ColorVector)→ColorVector : ⟨**function**⟩$_2$ }

Then the behavior application *B_Plus*(aVector,aVector)[10] will dispatch to the function ⟨**function**⟩$_1$ as will the applications *B_Plus*(aColorVector,aVector) and *B_Plus*(aVector,aColorVector). In all three cases, we will expect the return type to be `Vector`. However, the behavior application *B_Plus*(aColorVector,aColorVector) will be dispatched to ⟨**function**⟩$_2$ and the expected result type will be `ColorVector`. This example shows that our framework is designed for languages with *multiple dispatch*. This is the reason we allow covariant argument specifications. It also shows that our toy language is capable of correcty typing binary methods which is a known problem for many of object-oriented type systems ([AC95], [BCC+96]).

The function definitions of our toy language are straightforward and precisely model lambda abstractions. The `let`-construct is just a convenient way to introduce local names. It does not play any special role in terms of typing (as does, for example, the polymorphic let construct of various languages in ML family [MTH90], [Car86]). We will use a syntactic sugar **fun** ($x_1, \ldots, x_n$) ⟨**expr**⟩ for

fun (x)
    let x_1 = *B_Project*$_1$(x) **in**
    ⋮
    let x_n = *B_Project*$_n$(x) **in expr**

in order to deal with multiple arguments.

A non-trivial construct in our toy language is ⟨**class**⟩ which is syntactically equivalent to **class**(x, ⟨TS⟩), where x is a name (introduced by ⟨**let**⟩ or ⟨**fun**⟩) or a constant. The meaning of it can be described as follows: the run-time type of the object referred to by x is taken (let it be x) and tranformed by the type f derived from the type specification ⟨TS⟩ to yield a type t. Then, the object class(t) is returned ($t = \text{apply}(f, x)\Downarrow$). To illustrate how this works, consider the simplified version of this construct **cclass**(⟨TS⟩) which is a syntactic sugar for **class**(**unit**, Unit→⟨TS⟩). For example, **cclass**(Person) will produce the object class(Person) that can be used to create new persons as in

let newPerson = *B_New*(**cclass**(Person)) **in** ...

A more involved example is a function that swaps values of two variables of the same type:

fun (x,y)

[10] We will use ⟨expr⟩(⟨expr⟩$_1$, ..., ⟨expr⟩$_n$) as a syntactic sugar for ⟨expr⟩(⟨⟨expr⟩$_1$, ..., ⟨expr⟩$_n$⟩).

```
⟨program⟩ ::= ⟨user-type-decls⟩ ; ⟨behaviors⟩ ; ⟨expr⟩
⟨user-type-decls⟩ ::= ⟨user-type-decl⟩
      | ⟨user-type-decls⟩ ; ⟨user-type-decl⟩
⟨user-type-decl⟩ ::= ⟨type-decl⟩
      | ⟨subtype-decl⟩
⟨behaviors⟩ ::= ⟨behavior⟩
      | ⟨behaviors⟩ ; ⟨behavior⟩
⟨behavior⟩ ::= behavior ⟨name⟩ { ⟨assoc-list⟩ }
⟨assoc-list⟩ ::= ⟨assoc⟩
      | ⟨assoc-list⟩ , ⟨assoc⟩
⟨assoc⟩ ::= ⟨TS⟩ : ⟨gen-function⟩
⟨gen-function⟩ ::= ⟨function⟩
      | primitive-name

⟨function⟩ ::= fun (x) ⟨expr⟩
⟨expr⟩ ::= x (x is a variable name)
      | c (c is a constant name)
      | ⟨class⟩
      | ⟨function⟩
      | ⟨applic⟩
      | ⟨let⟩
      | ⟨product⟩
      | ( ⟨exprs⟩ )
⟨applic⟩ ::= ⟨expr⟩(⟨expr⟩)
⟨product⟩ ::= ⟨⟨expr⟩, . . . , ⟨expr⟩⟩
⟨let⟩ ::= let x = ⟨expr⟩ in ⟨expr⟩
⟨class⟩ ::= class(x,⟨TS⟩)
⟨exprs⟩ ::= ⟨expr⟩
      | ⟨exprs⟩ ; ⟨expr⟩
```

Figure 4: Toy language syntax

let temp $=$ *B_New*(class(x,X$\rightarrow$X)) in (*B_set*(temp,x); *B_set*(x,y); *B_set*(y,temp))
This creates a temporary variable of the same type as the first argument and uses it as a temporary storage. Of course, this could be done simpler by

```
fun (x,y)
    let temp = x in (B_set(x,y); B_set(y,temp))
```

since $\langle$let$\rangle$ is eagerly evaluated.

The $\langle$class$\rangle$ construct in our toy language is just an example of a kind of language constructs that are posssible in our framework. The ability not only to extract, but also manipulate types at run-time is inherent in the framework presented in this paper.

The remaining constructs are standard and staightforward and do not require special description.

The typing rules for the toy language are presented in the Figure 5. Θ is *the typing environment* which is the set of name to type associations of the form $\mathtt{x} : t$ understood as "the name x is known to correspond to an object with a type that is a subtype of t or t itself".

Top-level behavior definitions and predefined constant names along with their types constitute the *initial typing environment* (denoted Θ_0) that will be formally defined in the next section.

$$\frac{\Theta, \mathtt{x} : x \triangleright \langle\mathtt{expr}\rangle : e}{\Theta \triangleright \mathtt{fun}\ (\mathtt{x})\ \langle\mathtt{expr}\rangle : \lambda x.e}\ \mathsf{Fun}$$

$$\frac{\Theta \triangleright \langle\mathtt{exprs}\rangle : e}{\Theta \triangleright (\langle\mathtt{exprs}\rangle) : e}\ \mathsf{Block}$$

$$\frac{\Theta \triangleright \langle\mathtt{exprs}\rangle : e_1 \qquad \Theta \triangleright \langle\mathtt{expr}\rangle : e_2}{\Theta \triangleright \langle\mathtt{exprs}\rangle ; \langle\mathtt{expr}\rangle : \mathrm{cond}(\{e_1 \le \mathtt{Object}\}); e_2}\ \mathsf{Seq}$$

$$\frac{\Theta \triangleright \langle\mathtt{expr}\rangle_1 : e_1 \qquad \Theta, \mathtt{x} : e_1 \triangleright \langle\mathtt{expr}\rangle_2 : e_2}{\Theta \triangleright \mathtt{let}\ \mathtt{x} = \langle\mathtt{expr}\rangle_1\ \mathtt{in}\ \langle\mathtt{expr}\rangle_2 : e_2}\ \mathsf{Let}$$

$$\frac{\Theta \triangleright \langle\mathtt{expr}\rangle_1 : e_1 \qquad \Theta \triangleright \langle\mathtt{expr}\rangle_2 : e_2}{\Theta \triangleright \langle\mathtt{expr}\rangle_1(\langle\mathtt{expr}\rangle_2) : \mathrm{apply}(e_1, e_2)}\ \mathsf{Appl}$$

$$\frac{\Theta \triangleright \langle\mathtt{expr}\rangle_1 : e_1 \qquad \dots \qquad \Theta \triangleright \langle\mathtt{expr}\rangle_n : e_n}{\Theta \triangleright \langle\langle\mathtt{expr}\rangle_1, \dots, \langle\mathtt{expr}\rangle_n\rangle : \mathtt{Product}_n(e_1, \dots, e_n)}\ \mathsf{Prod}$$

$$\frac{\Theta \triangleright \mathtt{x} : x \qquad \langle\mathtt{TS}\rangle \mapsto^+ S \qquad \vdash S : s \qquad \mathrm{apply}(s, x)\Downarrow = t \qquad t \le_c \mathtt{RegObject}}{\Theta \triangleright \mathtt{class}(\mathtt{x}, \langle\mathtt{TS}\rangle) : \mathtt{Class}(t)}\ \mathsf{Class}$$

where x is either a variable or a constant name

$$\frac{}{\Theta, \mathtt{u} : t \triangleright \mathtt{u} : t}\ \mathsf{Axiom}$$

Figure 5: Typing rules for the toy language

In order to define natural semantics for the toy language, we will need to define what can be a run-time object and what is the run-time environment.

A run-time objects in the language is one of

1. Predefined constant c_i
2. Behavior $\mathsf{b} = \{\langle \texttt{TS} \rangle_i : \mathsf{f}_i\}$
3. Closure $\mathsf{closure}(E, \mathtt{x}, \langle \texttt{expr} \rangle)$
4. Primitive-closure $\mathsf{primitive}_i$
5. Run-time object o
6. Class object $\mathsf{class}(t)$
7. The unit value unit (used for command results, procedure results etc)
8. Type error $\mathsf{err}^{\top}$
9. Run-time error $\mathsf{err}^{\perp}$

The behavior objects thus carry enough information to be able to deal with run-time dispatch (late binding). Closures are produced by evaluation of function definitions ($\langle$**function**$\rangle$) and carry the environment that was in effect when the function definition was evaluated, the abstraction variable, and the body. Primitive-closures are analogs of closures for primitive functions. They do not need to keep the environment as primitive functions can not access it. Class objects are objects of the types **Class**(+) and serve as factories of objects. The run-time type error $\mathsf{err}^{\top}$ (of type $\top$) should never be produced in a type-checked program, while a run-time error $\mathsf{err}^{\perp}$ (of type $\perp$) can be produced by a primitive function and serves as an exception because it is pusehed up until the top level is reached.

The run-time environment E is a list of associations of the form $\mathtt{x} = \mathsf{o} : t$ where $\mathtt{x}$ is a name, o is an object, and t is its type. Thus we maintain types of objects at run-time.

The reduction ($\bigtriangledown$) is defined on triples $(S, E, \langle \texttt{expr} \rangle)$ (where S is store and E is an environment) and produces tuples of the form $(S', \mathsf{o} : t)$ where S' is a new store, o is a run-time object, and t is its type. The reduction rules are presented in the Appendix F.

The decision to adopt an eager evaluation for behavior and function arguments has been dictated by the necessity to perform run-time dispatch. Until an argument is not evaluated, the dispatch decision can not be made and the function to execute can not be chosen. Another construct that requires run-time type information is the $\langle$**class**$\rangle$ construct discussed earlier.

Note that function abstractions are evaluated lazily. This allows us to use the behavior definition mechanism to define *commands* in the language instead of making them a part of the kernel. In the following example, we will define the command *B_if-command* assuming that we have a type **Boolean** and a behavior *B_if* with the fillowing definitions:

type Boolean;
behavior *B_if* { (**Boolean**, X, Y)→(X ∪Y) : **primitive-if** };
The command version will take a boolean and two commands and produce a command.
behavior *B_if-command* {
(**Boolean**, **Unit** →**Unit**, **Unit** →**Unit**)→(**Unit** →**Unit**)) : **fun** (**x**,**y**,**z**) *B_if*(**x**,**y**,**z**) };

Here the second and the third argument are closures which are passed along and one of them constitutes the result. The type **Unit**→**Unit** is a generic type of commands. The language operator ';' could have been defined this way as well, in which case we would have one less basic construct in the language.

In this section, we have introduced the toy language, its syntax, semantics, and typing rules. In the next section, we will deal with issues of behavior consistency and type correctness, and present the subject reduction theorem.

7 Behavior consistency and type-checking

In this section, we will consider *behavior consistency* issues and present the subject reduction theorem for the toy language.

Behavior consistency is a sum of three components: the function association consistency, the primitive function association consistency, and the unambiguous choice of a behavior association during dispatch. The function association consistency ensures that a function in an association does indeed conform to the type specification of the association. This corresponds to the type correctness of the function with respect to a given type specification. The primitive function consistency ensures the conformance of primitive functions.

The unambiguous choice of a behavior association is needed to ensure that the dispatch can always pick "the best fit" association based on the types of actual arguments. For example, if we have a type `Person` and its subtypes `Student` and `Teacher`, we can define a behavior *B_changeDepartment*:

`behavior` *B_changeDepartment* {
    (`Person`,`Department`)→`Unit` : ⟨`function`⟩$_1$,
    (`Student`,`Department`)→`Unit` : ⟨`function`⟩$_2$,
    (`Teacher`,`Department`)→`Unit` : ⟨`function`⟩$_3$ }

which invokes different functions for different types. It is easy to pick the "best fit" association in this case. However, if we add a new type, `TeachingAssistant`, which is a subtype of both `Student` and `Teacher`, the behavior definition above will become ambiguous as it is not clear which function should be applied when the following statement is executed: *B_changeDepartment*(aTeachingAssistant).

In order to deal with situations like the one described above, we introduce the operator $\cap_?$ which, being applied to two type specifications, yields a set of extended argument types that can potentially conform to both type specifications. In the example above, `Student` $\cap_?$ `Teacher` $= \emptyset$ before the introduction of the type `TeachingAssistant` and `Student` $\cap_?$ `Teacher` $=$ {`TeachingAssistant`} after that. The rules for the operator $\cap_?$ are presented in the Figure 6 and Figure 7.

The main property of the operator $\cap_?$ defined above is given by the following theorem:

Theorem 7.1. *Domain intersection:* If a_1, a_2 are argument types, t is a concrete type, and there exist such concrete types $\vec{t''}$ and $\vec{t'}$ that $t \leq_f [\vec{t'}/\vec{x}_1]a_1(t')$ and $t \leq_f [\vec{t''}/\vec{x}_2]a_2$, then

$$\exists \hat{t} \in [\vec{*}/\vec{x}_1]a_1 \cap_? [\vec{*}/\vec{x}_2]a_2, \vec{ac}: \quad t \leq_f [\vec{ac}/\vec{*}]\hat{t}$$

where $\hat{t}$ is an extended reduced type, $\vec{ac}$ are concrete types, and $\vec{x}_i$ denotes free variables of a_i. □

Sketch of the proof: The proof is by structural induction. For each rule we show that if the statement of the theorem is true for the premises of the rule, then it is also true for that rule's conclusions. We use the analogous statements with $\leq_f$ changed to $\geq_f$ and $=_f$ respectively when dealing with rules about $\cup_?$ and $\equiv_?$. Note that we do not need $\equiv_?$ and $\cup_?$ rules for arrow types as they are guaranteed not to occur in other than covariant positions. The most tedious part of the proof is the one dealing with the constructor rules which are defined in terms of functions $\searrow$ and $\nearrow$ over $\mathcal{G}$.

In other words, a non-empty intersection produced by the operator $\cap_?$ does not guarantee that there is a concrete type in the intersection; however, if such a type t exists, the intersection is not empty and t conforms to one of the types in the intersection.

Now we are ready to formally define behavior consistency.

Definition 7.1. *Behavior consistency:* A behavior is *consistent* iff

$$\frac{a \cap_? c = C_1 \qquad b \cap_? c = C_2}{a \cup b \cap_? c = C_1 \cup C_2}\ \mathsf{DI}\cup \qquad \frac{a \cup_? c = C_1 \qquad b \cup_? c = C_2}{a \cup b \cup_? c = \cup_{\substack{c_1 \in C_1 \\ c_2 \in C_2}} (c_1 \cup_? c_2)}\ \mathsf{DU}\cup$$

$$\frac{a \equiv_? b = C_1}{a \cup b \equiv_? c = \cup_{c_1 \in C_1} (c_1 \equiv_? c)}\ \mathsf{DE}\cup$$

$$\frac{a \cap_? c = C_1 \qquad b \cap_? c = C_2}{a \cap b \cap_? c = \cup_{\substack{c_1 \in C_1 \\ c_2 \in C_2}} (c_1 \cap_? c_2)}\ \mathsf{DI}\cap \qquad \frac{a \cup_? c = C_1 \qquad b \cup_? c = C_2}{a \cap b \cup_? c = C_1 \cup C_2}\ \mathsf{DU}\cap$$

$$\frac{a \equiv_? b = C_1}{a \cap b \equiv_? c = \cup_{c_1 \in C_1} (c_1 \equiv_? c)}\ \mathsf{DE}\cap$$

$$\frac{}{a \rightarrow b \cap_? c \rightarrow d = \{(a \rightarrow b) \cap (c \rightarrow d)\}}\ \mathsf{DI}\rightarrow$$

$$\frac{}{a \rightarrow b \cap_? \texttt{Behavior} = \{a \rightarrow b\}}\ \mathsf{DI}\texttt{Behavior}$$

$$\frac{}{a \cap_? \texttt{Object} = \{a\}}\ \mathsf{DI}\texttt{Object} \qquad \frac{}{a \cup_? \texttt{Object} = \{\texttt{Object}\}}\ \mathsf{DU}\texttt{Object}$$

$$\frac{}{a \cap_? * = \{a\}}\ \mathsf{DI*} \qquad \frac{}{a \cup_? * = \{a\}}\ \mathsf{DU*} \qquad \frac{}{a \equiv_? * = \{a\}}\ \mathsf{DE*}$$

$$\frac{\{a_i \cap_?^{V_{P,i}} b_i = C_i\}_{i=1,\ldots,n_P}}{p(a_1,\ldots,a_{n_P}) \cap_? p(b_1,\ldots,b_{n_P}) = \cup_{c_i \in C_i}(p(c_1,\ldots,c_{n_P})}\ \mathsf{DIP}=$$

$$\frac{\{a_i \cup_?^{V_{P,i}} b_i = C_i\}_{i=1,\ldots,n_P}}{p(a_1,\ldots,a_{n_P}) \cup_? p(b_1,\ldots,b_{n_P}) = \cup_{c_i \in C_i}(p(c_1,\ldots,c_{n_P})}\ \mathsf{DUP}=$$

$$\frac{\{a_i \equiv_? b_i = C_i\}_{i=1,\ldots,n_P}}{p(a_1,\ldots,a_{n_P}) \equiv_? p(b_1,\ldots,b_{n_P}) = \cup_{c_i \in C_i}(p(c_1,\ldots,c_{n_P})}\ \mathsf{DEP}=$$

Here $\cap_?{}^0 = \cup_?{}^0 = \equiv_?$, $\cap_?{}^+ = \cup_?{}^- = \cap_?$, and $\cup_?{}^+ = \cap_?{}^- = \cup_?$.

$$\frac{b \cap_? a = C}{a \cap_? b = C}\ \mathsf{DISimm} \qquad \frac{b \cup_? a = C}{a \cup_? b = C}\ \mathsf{DUSimm} \qquad \frac{b \equiv_? a = C}{a \equiv_? b = C}\ \mathsf{DESimm}$$

$$\frac{}{a \cap_? a = \{a\}}\ \mathsf{DIRefl} \qquad \frac{}{a \cup_? a = \{a\}}\ \mathsf{DURefl} \qquad \frac{}{a \equiv_? a = \{a\}}\ \mathsf{DERefl}$$

Figure 6: Domain intersection

$$\frac{\begin{array}{c}\text{For all } S_i \in \mathcal{G}: \quad S_i \leq_{\mathcal{G}} P \wedge S_i \leq_{\mathcal{G}} P' \wedge \\ ((S_i \leq_{\mathcal{G}} S' \wedge S' \leq_{\mathcal{G}} P \wedge S' \leq_{\mathcal{G}} P') \Rightarrow S' = S_i) \\ p(a_1, \ldots, a_{n_P}) \searrow s_i = C_1^i \qquad p'(b_1, \ldots, b_{n'_P}) \searrow s_i = C_2^i\end{array}}{p(a_1, \ldots, a_{n_P}) \cap_? p'(b_1, \ldots, b_{n'_P}) = \cup_{i,\, c_1^i \in C_1^i,\, c_2^i \in C_2^i} (c_1^i \cap_? c_2^i)} \mathsf{DIP}{\neq}$$

The rule DIP$\neq$ is also in effect when there is t on either side of $\cap_?$.

$$\frac{\begin{array}{c}\text{For all } S_i \in \mathcal{G}: \quad P \leq_{\mathcal{G}} S_i \wedge P' \leq_{\mathcal{G}} S_i \wedge \\ ((S' \leq_{\mathcal{G}} S_i \wedge P \leq_{\mathcal{G}} S' \wedge P' \leq_{\mathcal{G}} S') \Rightarrow S' = S_i) \\ p(a_1, \ldots, a_{n_P}) \nearrow s_i = C_1^i \qquad p'(b_1, \ldots, b_{n'_P}) \nearrow s_i = C_2^i\end{array}}{p(a_1, \ldots, a_{n_P}) \cup_? p'(b_1, \ldots, b_{n'_P}) = \cup_{i,\, c_1^i \in C_1^i,\, c_2^i \in C_2^i} (c_1^i \cup_? c_2^i)} \mathsf{DUP}{\neq}$$

The rule DUP$\neq$ is also in effect when there is t on either side of $\cup_?$.

$$\frac{P(x) \leq_u x \qquad a \neq p(c) \qquad a \cup_? b = C}{a \cup_? p(b) = C} \mathsf{DUP}_{any}$$

Figure 7: Domain intersection: parametric types

1. Every type specification of the behavior is consistent
2. Every association of the behavior is consistent
3. Choice of an association is unambiguous

□

Definition 7.2. *Type specification consistency:* A type specification ⟨TS⟩ is *consistent* iff

1. $\langle \texttt{TS} \rangle \mapsto^{+} S'$
2. $\vdash s :: S/C$
3. For every $\{a_1 \leq^v a_2\} \in C$: $\quad a_1 \Downarrow \leq_c^v a_2 \Downarrow$

□

The meaning of this definition is that the type specification must be provable by the type specification logic and all top-level constraints in that proof must be satisfied.

Definition 7.3. *Function association consistency:* An association ⟨TS⟩ : ⟨function⟩ is *consistent* iff

1. $\Theta_0 \triangleright \langle \texttt{function} \rangle : f$
2. $\vdash_{TR} t :: \langle \texttt{TS} \rangle$
3. $f \leq_c t$

Here Θ_0 is *the initial typing environment* defined next. □

Note that we use the "naive" logical derivation ($\vdash_{TR}$) instead of a full-fledged logic ($\vdash$). We have to do that to be able to use computable subtyping ($\leq_c$) which does not handle lambda abstractions on the right unless they are arrow types. The naive translation is guaranteed to produce only arrow types on the right. Consider the association
X →X : `fun (x) x`
In this case, $f = \lambda x.x$ and $t = x' \rightarrow x'$. In order to prove $f \leq_c t$ we have to prove $\text{apply}(f, x')\Downarrow \leq_c x'$ (according to the λ-rule of computable subtyping). But $\text{apply}(f, x')\Downarrow = x'$ and $x' \leq_c x'$. Thus, the association above is consistent.

The *initial typing environment* (denoted Θ_0) is a typing environment that includes types of all constants defined at the top level of the program. That includes predefined constants and behaviors.

Definition 7.4. *Initial typing environment:* Θ_0 is the initial typing environment iff Θ_0 consists of:

1. $\{\texttt{unit} : \texttt{Unit}\}$ (typing of `unit`)
2. $\{\texttt{c}_i : c_i\}$ for all predefined constants $\texttt{c}_i$
3. $\{\texttt{b}_i : bt_i\}$ for all behaviors $\texttt{b}_i$, where bt_i are defined as follows:

$$bt = \cap_{j=1}^{n_b} s_j, \ \ \langle\texttt{TS}\rangle_j \mapsto^+ S'_j, \ \ \vdash s_j :: S'_j/C$$

Here $\langle\texttt{TS}\rangle_j$ $(j = 1, \ldots, n_b)$ are type specifications of b.

□

Thus a type of a behavior that has several association is the lower bound of types derived from all type specifications for that behavior. For example, the type of behavior *B_Add* defined as
`behavior` *B_Add* {
    (`Real`,`Real`) →`Real` : ⟨`function`⟩$_1$,
    (`Integer`,`Integer`) →`Integer` : ⟨`function`⟩$_2$
is
$(\lambda x.(\text{cond}(x{\downarrow}1 \leq \texttt{Real}, x{\downarrow}2 \leq \texttt{Real}); \texttt{Real}))\cap$
    $(\lambda x.(\text{cond}(x{\downarrow}1 \leq \texttt{Integer}, x{\downarrow}2 \leq \texttt{Integer}); \texttt{Integer}))$
If we apply it to an argument type (`Ineger`,`Real`) we will get $\texttt{Real}\cap\top = \texttt{Real}$. If an argument type was (`Ineger`,`Integer`), we would get $\texttt{Real}\cap\texttt{Integer} = \texttt{Integer}$ (assuming `Integer` is a subtype of `Real`).

The following definition deals with unambiguous choice of an association from the list of associations of a particular behavior.

Definition 7.5. *Unambiguous choice of association:* We say that *the choice of association for a behavior b is unambiguous* if for any pair of type specifications of b $\langle\texttt{TS}\rangle_1$ and $\langle\texttt{TS}\rangle_2$ the following holds:

$$\begin{array}{l}\forall t \in arg_1 \cap_? arg_2 \ \ \exists! i: \\ \quad \langle\texttt{TS}\rangle_i \leq_{\text{cov}} \langle\texttt{TS}\rangle_1 \wedge \langle\texttt{TS}\rangle_i \leq_{\text{cov}} \langle\texttt{TS}\rangle_2 \wedge \text{apply}(f_i, t')\Downarrow \leq_c \texttt{Object}\end{array}$$

where

$$arg_1 = [\vec{*}/\vec{x}]tr_1 \ \ (\vdash_{TR} tr_1 :: Arg_1, \ \ \langle\texttt{TS}\rangle_1 = Arg_1 \rightarrow Res_1)$$
$$arg_2 = [\vec{*}/\vec{x}]tr_2 \ \ (\vdash_{TR} tr_2 :: Arg_2, \ \ \langle\texttt{TS}\rangle_2 = Arg_2 \rightarrow Res_2)$$
$$\langle\texttt{TS}\rangle_i \mapsto^+ S_i, \ \ \vdash f_i :: S_i/C_i \ \ t' = [\vec{y}/\vec{*}]t$$

Here $[\vec{y}/\vec{*}]t$ denotes t with every occurence of $*$ changed to a fresh variable y_i, while $[\vec{*}/\vec{x}]tr_i$ denotes tr_i with all free variables x_i changed to $*$. □

This definition ensures that for any type-correct argument types for the behavior b there is always one association with a type specification that covers all others to which the argument types conform. The definition of covering is given below.

Definition 7.6. *Cover:* A type specification $\langle \mathtt{TS} \rangle_1$ *covers* another type specification $\langle \mathtt{TS} \rangle_2$ (denoted $\langle \mathtt{TS} \rangle_2 \leq_{\mathrm{cov}} \langle \mathtt{TS} \rangle_1$) iff

1. $\vdash_{TR} a_1 \to r_1 :: \langle \mathtt{TS} \rangle_1$
2. $\vdash_{TR} a_2 \to r_2 :: \langle \mathtt{TS} \rangle_2$
3. $\langle \mathtt{TS} \rangle_1 \mapsto^+ S_1, \qquad \vdash s_1 :: S_1/C$
4. $r_2 \leq_c r_1$
5. $\mathrm{apply}(s_1, a_2)\Downarrow \leq_c \mathtt{Object}$

□

Informally, a type specification $A \to B$ covers another one $A' \to B'$ if $A \geq A'$ and $B \geq B'$. We can say that the second type specification is more specific than the first one as it specifies both argument and result types more precisely. For example, the type specification `Real`→`Real` covers the type specification `Integer`→`Integer`.

Now we are ready to define dispatch. The goal of *dispatching* a behavior b on an argument of (concrete) type t is to pick the most specific association for t from the list of associations of b and apply the function from this association to the argument.

Definition 7.7. *Dispatch:* The algorithm for picking up the most specific association ($dispatch(b,t)$) is defined as follows:

1. Let $\langle \mathtt{TS} \rangle_i$ be all associations of b, $\langle \mathtt{TS} \rangle_i \mapsto^+ S_i$, $\vdash s_i :: S_i/C$
2. We form the set $M = \{i \mid \quad \mathrm{apply}(s_i, t) \leq_c \mathtt{Object}\}$
3. The set M' is formed by all $i' \in M$ such that $\langle \mathtt{TS} \rangle_{i'}$ is not covered by any other $\langle \mathtt{TS} \rangle_i$, $i \in M$. If $|M'| = 1$ we say that the dispatch on b and t is *consistent and unambiguous* and pick the association $i', i' \in M'$ as the most specific one: $dispatch(b,t) = i'$. If, on the other hand, $|M'| \neq 1$, we put $dispatch(b,t) = 0$.

□

M is the set of the specifications to which the argument type conforms. From those, we pick the most specific one.

The following theorem establishes that once a behavior is consistent, dispatch is always unambiguous. Moreover, the function chosen by dispatch produces the least possible result type.

Theorem 7.2. *Behavior consistency:* If behavior b is consistent, $b : tb = \cap_{i=1,\dots,n} s_i \in \Theta_0$, and t is a concrete type such that $\mathrm{apply}(tb, t)\Downarrow \leq_f \mathtt{Object}$, then

1. $dispatch(b,t) \neq 0$

2. If $dispatch(b,t) = i$ then

$$\text{apply}(s_i, t)\Downarrow =_f \text{apply}(tb, t)\Downarrow$$

3. If $dispatch(b,t) = i$, function $\langle\texttt{function}\rangle_i$ is a part of this association, and $\Theta_0 \triangleright \langle\texttt{function}\rangle_i : f_i$, then

$$\text{apply}(f_i, t)\Downarrow \leq_f \text{apply}(tb, t)\Downarrow$$

□

The proof of this theorem is given in the Appendix G. This proof is indicative of the techniques we use to prove the other theorems in this section.

So far we have considered the functions written in our toy language. The following definition establishes the properties of primitive functions that are necessary for the type-checking and dispatch to work properly.

Definition 7.8. *Primitive function association consistency:* A primitive function association $\langle\texttt{TS}\rangle : \texttt{primitive}_i$ is *consistent* if for any consistent store S the following holds:
If

$$\begin{array}{c} (S, \mathsf{o} : t)\nabla_{prim_i} = (S', \mathsf{o}' : t') \\ \langle\texttt{TS}\rangle \mapsto^+ S \qquad \vdash s : S/C \\ \text{apply}(s,t)\Downarrow \leq_f \texttt{Object} \end{array}$$

then

$$\begin{array}{c} t' \leq_f \text{apply}(s,t)\Downarrow \\ S' \text{ is consistent} \end{array}$$

□

In other words, we require that primitive functions behave almost as good as type-checked non-primitive ones w.r.t their type specifications. Almost as good because we do not require monotonicity of typing, just conformance to the type specification. Note that there is no definition of store consistency here as it depends upon the semantics of primitive functions. However, for the Theorem 7.3 the concrete definition of store consistency is unnecessary as long as correct applications of primitive functions do not disturb it (as stated in the above definition).

Now we have all the components needed for the subject reduction theorem.

Theorem 7.3. *Subject reduction:* If

1. The user type graph $\mathcal{G}$ is consistent
2. All behaviors are consistent
3. All primitive function associations are consistent
4. Store S is consistent
5. All free object variables $\mathbf{x}_i$ in $\langle\texttt{expr}\rangle$ are present in E

6. $E = \{\mathtt{x}_i = \mathtt{o}_i : t_i\}$, $\Theta = \{\mathtt{x}_i : t_i\}$, $\Theta \triangleright \langle\mathtt{expr}\rangle : t_e$, and $t_e \Downarrow \leq_f$ `Object`
7. $(S, E, \langle\mathtt{expr}\rangle)\bigtriangledown = (S', \mathtt{o} : q)$

then

1. S' is consistent
2. $q \leq_f t_e\Downarrow$

□

The consequence of this theorem is that the reduction of type-correct terms does not produce run-time type errors ($\mathtt{err}^\top$) or inconsistent store. Note that other run-time errors ($\mathtt{err}^\perp$) can only be produced by primitive functions, if they are so defined.

Sketch of the proof: The proof is by structural induction on $\bigtriangledown$. The most involved case is the one dealing with application $(\langle\mathtt{expr}\rangle_1(\langle\mathtt{expr}\rangle_2))$ when neither of the expressions is reduced to $\mathtt{err}^\perp$. This case in considered in the Appendix G

The absence of type errors in the correctly type-checked program is established as a consequence of the subject reduction theorem.

Corollary 7.1. Type correctness of a program If $\langle\mathtt{program}\rangle = \langle\mathtt{user\text{-}type\text{-}specs}\rangle\langle\mathtt{behaviors}\rangle\langle\mathtt{expr}\rangle$ is a program and

1. The user type graph $\mathcal{G}$ generated by type and subtype definitions in the program is conssitent
2. All behaviors in the program are consistent
3. All primitive function associations in the program are consistent
4. S_0 is a consistent store
5. Θ_0 is the initial typeing environment (Definition 7.4)
6. $E_0 = \{\mathtt{unit} = \mathtt{unit} : \mathtt{Unit}, \mathtt{b}_i = \mathtt{b}_i : bt_i, \mathtt{c}_i = \mathtt{c}_i : c_i\}$ where $\mathtt{c}_i$ are predefined constants and $\mathtt{b}_i$ are behaviors declared in the program. bt_i are behavior types defined as in Θ_0
7. $\Theta_0 \triangleright \langle\mathtt{expr}\rangle : p$
8. $p \leq_c$ `Object`
9. $(S_0, E_0, \langle\mathtt{expr}\rangle)\bigtriangledown = (S', \mathtt{o} : t_r)$

then

1. $t_r \leq_c$ `Object`
2. S' is consistent

□

In this section, we have introduced the notion of behavior consistency and dispatch. We have shown that consistent behaviors are always dispatched unambiguously. We have also presented the subject reduction theorem for our toy language and have established the absence of run-time type errors in a successfully type-checked program.

8 Conclusions

In this paper, we have presented a type system that combines parametric and inclusion polymorphism, variance specifications, user-defined subtyping between parametric and ordinary types, precise function types, multiple dispatch, and static typing. We have shown that our approach based on the type specification logic allows us to correctly type-check programs in the toy language for which we have proven the subject-reduction theorem.

In addition to the features described above, the presented type system has lower and upper bound types and is therefore capable of typing database queries. The ability to specify subtype relationships of the form $\texttt{Var}(X) \leq X$ allows us to treat imperative features such as assignment, variables, and creation of new objects inside our framework.

Further research directions include the development of less strict conditions for behavior consistency, lifting some of the remaining restrictions on subtype declarations, enhancement of the toy language (the ability to define behaviors at run-time, run-time operations on types), and building of its prototype compiler.

References

[AC95] Martin Abadi and Luca Cardelli, *On subtyping and matching*, Proceedings ECOOP'95, 1995, pp. 145–167.

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay, *Static type checking of multi-methods*, ACM SIGPLAN Notices **26** (1991), no. 11, 113–128, In Proc. of OOPSLA'91.

[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** (1996), no. 3, 221–242.

[BO96] P. Buneman and A. Ohori, *Polymorphism and type inference in database programming*, ACM Transactions on Database Systems **21** (1996), no. 1, 30–76.

[BSG95] Kim B. Bruce, Angela Schuett, and Robert Van Gent, *PolyTOIL: A type-safe polymorphic object-oriented language*, Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95) (Åarhus, Denmark) (W. Olthoff, ed.), LNCS 952. Springer, August 1995, Extended abstract.

[Car86] Luca Cardelli, *Amber*, Combinators and Functional Programming Languages (G. Cousineau, P-L. Curien, and B. Robinet, eds.), Springer-Verlag, 1986, LNCS 242.

[CL94] Craig Chambers and Gary T. Leavens, *Typechecking and modules for multi-methods*, ACM SIGPLAN Notices **29** (1994), no. 10, 1–15, In Proc. of OOPSLA'94.

[CL96] Craig Chambers and Gary T. Leavens, *BeCecil, A core object-oriented language with block structure and multimethods: Semantics and typing*, Tech. Report 96-17, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, December 1996, Available by anonymous ftp from ftp.cs.iastate.edu, and by e-mail from almanac@cs.iastate.edu. Also University of Washington Department of Computer Science and Engineering TR number UW-CSE-96-12-02.
URL: `ftp://ftp.cs.iastate.edu/pub/techreports/TR96-17/TR.ps.Z`

[DGLM95] M. Day, R. Gruber, B. Liskov, and A. C. Myers, *Subtypes vs where clauses: Constraining parametric polymorphism*, SIGPLAN Notices **30** (1995), no. 10, 156–168.

[Dyc92] Roy Dyckhoff, *Contraction-free sequent calculi for intuitionistic logic*, The Journal of Symbolic Logic **57** (1992), no. 3, 795–807.

[ESTZ95] Jonathan Eifrig, Scott Smith, Valery Trifonov, and Amy Zwarico, *An interpretation of typed OOP in a language with state*, LISP and Symbolic Computation (1995), no. 8, 357–397.

[Ghe91] Giorgio Ghelli, *A static type system for message passing*, ACM SIGPLAN Notices **26** (1991), no. 11, 129–145, In Proc. of OOPSLA'91.

[GM96] Andreas Gawecki and Florian Matthes, *Integrating subtyping, matching and type quantification: A practical perspective*, Proceedings of the 10th European Conference on Object-Oriented Programming (Linz, Austria), Springer Verlag, July 1996.

[MBC+96] Ron Morrison, Fred Brown, Richard Connor, Quintin Cutts, Alan Dearle, Graham Kirbi, and Dave Munro, *Napier88 reference manual*, University of St. Andrews, March 1996, Release 2.2.1.

[Mey88] Bertrand Meyer, *Eiffel — the language*, Prentice-Hall, 1988.

[MMMP90] O. Matthes, B. Magnusson, and B. Moller-Pedersen, *Strong typing of object-oriented languages revisited*, ACM SIGPLAN Notices **25** (1990), no. 10, 140–150.

[MMS94] Florian Matthes, Sven Müßig, and J. W. Schmidt, *Persistent polymorphic programming in Tycoon: An introduction*, FIDE Technical Report Series FIDE/94/106, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.

[MTH90] Robin Miller, Mads Tofte, and Robert Harper, *The definition of Standard ML*, MIT Press, 1990.

[SOM93] Clemens Szypersky, Stephen Omohundro, and Stephan Murer, *Engineerig a programming language: The type and class system of Sather*, Tech. Report TR-93-064, The International Computer Science Institute, November 1993.

A Definition of conv, $\nearrow$ and $\searrow$

The functions defined here are auxiliary functions defined by user type specifications. conv converts a type to its supertype in the user type hierarchy during type computation. $\nearrow$ ($\searrow$) is a set-valued function that produces all possible conversions from subtype to supertype (from supertype to subtype) in the user type hierarchy. $\nearrow$ and $\searrow$ are used when domain intersection ($\cap_?$) is calculated.

Type-type If $T \leq_u T'$ then

$$\begin{aligned} \mathrm{conv}(t', t)\Downarrow &= t' \\ t \nearrow t' &= \{t'\} \\ t' \searrow t &= \{t\} \end{aligned}$$

Parametric-type If $P(c_1, \ldots, c_{n_P}) \leq_u T$ then

$$\begin{aligned} \mathrm{conv}(t, p(a_1, \ldots, a_{n_P}))\Downarrow &= (\mathrm{cond}(\{a_i \cap Max_{P,i} \leq^{V_{P,i}} c_i\}_{i=1,\ldots,n_P}); t)\Downarrow \\ p(a_1, \ldots, a_{n_P}) \nearrow t &= \{t\} \text{ if } (Max_{P,i} \cap a_i) \cap_?{}^{V_{P,i}}\ c_i \neq \emptyset \text{ for all } i = 1, \ldots, n_P \\ t \searrow p &= \{p(c_1, \ldots, c_{n_P})\} \end{aligned}$$

Type-parametric If $T \leq_u P(c_1, \ldots, c_{n_P})$ then

$$\begin{aligned} \mathrm{conv}(p, t)\Downarrow &= p(c_1, \ldots, c_{n_P}) \\ t \nearrow p &= \{p(c_1, \ldots, c_{n_P})\} \\ p(a_1, \ldots, a_{n_P}) \searrow t &= \{t\} \text{ if } (a_i \cap Max_{P,i}) \cap_?{}^{V_{P,i}}\ c_i \neq \emptyset \text{ for all } i = 1, \ldots, n_P \end{aligned}$$

Parametric-parametric If $P(x_1, \ldots, x_{n_P}) \leq_u P'(q_1(x_1, \ldots, x_{n_P}), \ldots, q_{n_{P'}}(x_1, \ldots, x_{n_P}))$ then

$$\begin{aligned} \mathrm{conv}(p', p(a_1, \ldots, a_{n_P}))\Downarrow &= (\mathrm{cond}(\{q_i(a_1 \cap Max_{P,1}, \ldots, a_{n_P} \cap Max_{P,n_P}) \leq Max_{P',i}\}_{i=1,\ldots,n_{P'}}); \\ &\quad p'(q_1(a_1 \cap Max_{P,1}, \ldots, a_{n_P} \cap Max_{P,n_P}), \ldots, \\ &\quad\quad q_{n_{P'}}(a_1 \cap Max_{P,1}, \ldots, a_{n_P} \cap Max_{P,n_P})))\Downarrow \\ p(a_1, \ldots, a_{n_P}) \nearrow p' &= \cup_{c_i \in C_i} \{p'(c_1, \ldots, c_i)\} \\ &\quad \text{where } C_i = q_i(a_1 \cap Max_{P,1}, \ldots, a_{n_P} \cap Max_{P,n_P}) \cap_? Max_{P',i} \\ p'(a_1, \ldots, a_{n'_P}) \searrow p &= \{p(s_1, \ldots, s_{n_P})\} \text{ where } s_i = \begin{cases} a_j \cap Max_{P,j} & \text{if } c_i = x_j \\ * & \text{otherwise} \end{cases} \\ &\quad \text{provided that } a_i \cap_? c_i \neq \emptyset \text{ for all } i\text{:} \quad c_i \neq x_k \end{aligned}$$

Parametric-any If $P(x) \leq_u x$ then

$$\begin{aligned} \mathrm{conv}(p', p(a))\Downarrow &= \mathrm{conv}(p', (a \cap Max_{P,1})\Downarrow)\Downarrow \text{ for all } P' \neq P \\ p(a) \nearrow p' &= \cup_{a_i \in A} a_i \nearrow p' \text{ where } A = a \cap_? Max_{P,1} \\ &\quad \text{for all } P' \neq P \\ a \searrow p &= \cup_{a_i \in A} \{p(a_i)\} \text{ where } A = a \cap_? Max_{P,1} \\ &\quad \text{provided that } a \neq p(b) \end{aligned}$$

Refl

$$\begin{aligned} \mathrm{conv}(p, p(a_1, \ldots, a_{n_P}))\Downarrow &= p(a_1, \ldots, a_{n_P}) \\ p(a_1, \ldots, a_{n_P}) \nearrow p &= \{p(a_1, \ldots, a_{n_P})\} \\ p(a_1, \ldots, a_{n_P}) \searrow p &= \{p(a_1, \ldots, a_{n_P})\} \end{aligned}$$

Trans If $\{\langle P = P_i^1, \ldots, P_i^{n_i} = P'\rangle\}_{i=1,\ldots,k}$ are all different paths from P to P' in $\mathcal{G}$, then

$$\begin{aligned}
\mathrm{conv}(p', p(a))\Downarrow &= (\cap_{i=1}^k(\mathrm{conv}(p_i^{n_i}, \mathrm{conv}(p_i^{n_i-1}, \ldots \mathrm{conv}(p_i^2, p(a))\Downarrow \ldots)\Downarrow)\Downarrow)\Downarrow \\
p(a)\nearrow p' &= \cup_{i=1}^k\{a_i^n\} \text{ where } a_i^{n_i} \in a_i^{n_i-1}\nearrow p_i^{n_i}, \ldots, a_i^2 \in p(a)\nearrow p_i^2 \\
p'(a)\searrow p &= \cup_{i=1}^k\{a_i^1\} \text{ where } a_i^1 \in a_i^2\searrow p_1, \ldots, a_i^{n_i-1} \in p'(a)\searrow p_i^{n-1}
\end{aligned}$$

Default If not defined by any of the above,

$$\begin{aligned}
\mathrm{conv}(p, a)\Downarrow &= \top \\
a\nearrow p &= \emptyset \\
a\searrow p &= \emptyset
\end{aligned}$$

B Variance annotation of type specifications

The annotation process works on type specifications and produces annotated type specifications.

$$\frac{}{T \mapsto^v T}\ \textsf{Const} \qquad \frac{}{X \mapsto^v X^v}\ \textsf{Var}$$

$$\frac{A \mapsto^v A' \qquad B \mapsto^v B'}{A \cup B \mapsto^v A' \cup B'}\ \textsf{Meet} \qquad \frac{A \mapsto^v A' \qquad B \mapsto^v B'}{A \cap B \mapsto^v A' \cap B'}\ \textsf{Join}$$

$$\frac{A \mapsto^+ A' \qquad B \mapsto^+ B'}{A \to B \mapsto^+ A' \to B'}\ \textsf{Arrow}$$

$$\frac{\{A_i \mapsto^{v \odot V_{P,i}} A_i'\}_{i=1,\ldots,n_P}}{P(A_1, \ldots, A_{n_P}) \mapsto^v P(A_1', \ldots, A_{n_P}')}\ \textsf{Param}$$

Here $v \odot 0 = 0 \odot v = 0, + \odot - = - \odot + = -$, and $+ \odot + = - \odot - = +$

C Type computations

Type computation process is defined for types and extended types.

$$t\Downarrow = t \qquad t \text{ is a constant}$$

$$x\Downarrow = x \qquad x \text{ is a variable}$$

$$*\Downarrow = *$$

$$\lambda x.f\Downarrow = \lambda x.f$$

$$(a \cap b)\Downarrow = \begin{cases} a\Downarrow & \text{if } a\Downarrow \leq_c b\Downarrow \\ b\Downarrow & \text{if } b\Downarrow \leq_c a\Downarrow \\ (a\Downarrow)\cap(\Downarrow b) & \text{otherwise} \end{cases}$$

$$(a \cup b)\Downarrow = \begin{cases} a\Downarrow & \text{if } b\Downarrow \leq_c a\Downarrow \\ b\Downarrow & \text{if } a\Downarrow \leq_c b\Downarrow \\ (a\Downarrow)\cup(\Downarrow b) & \text{otherwise} \end{cases}$$

$$\mathrm{apply}(a,b)\Downarrow = \begin{cases} [b/x]f\Downarrow & \text{if } a\Downarrow = \lambda x.f \\ (\mathrm{apply}(c,b)\Downarrow \cup \mathrm{apply}(d,b)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cup d \\ (\mathrm{apply}(c,b)\Downarrow \cap \mathrm{apply}(d,b)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cap d \\ \bot & \text{if } a\Downarrow = \bot \text{ and } b\Downarrow \leq_c \texttt{Object} \\ \top & \text{otherwise} \end{cases}$$

$$(\mathrm{cond}(C);a)\Downarrow = \begin{cases} a\Downarrow & \text{if } C = \{a_i \leq b_i\}_{i=1,\dots,n} \text{ and } (a_i\Downarrow) \leq_c (b_i\Downarrow) \text{ for all } i \text{ from 1 to } n \\ \top & \text{otherwise} \end{cases}$$

$$P(x_1,\dots,x_{n_P})\Downarrow = \begin{cases} P(x_1\Downarrow,\dots,x_{n_P}\Downarrow) & \text{if } (a_i\Downarrow) \leq_c \texttt{Object} \text{ for all } i \text{ from 1 to } n_P \\ \top & \text{otherwise} \end{cases}$$

$$\mathrm{rev}(p,i,a)\Downarrow = \begin{cases} b_i\Downarrow & \text{if } a\Downarrow = p'(a_1,\dots,a_{n_{P'}}) \\ & \quad \text{and } \mathrm{conv}(p,p'(a_1\Downarrow,\dots,a_{n_{P'}}\Downarrow))\Downarrow = p(b_1,\dots,b_{n_P}) \\ b_i\Downarrow & \text{if } a\Downarrow = t \text{ and } \mathrm{conv}(p,t)\Downarrow = p(b_1,\dots,b_{n_P}) \\ (\mathrm{rev}(p,i,c)\Downarrow \cup \mathrm{rev}(p,i,d)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cup d \text{ and } V_{P,i} = + \\ (\mathrm{rev}(p,i,c)\Downarrow \cap \mathrm{rev}(p,i,d)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cap d \text{ and } V_{P,i} = + \\ (\mathrm{rev}(p,i,c)\Downarrow \cap \mathrm{rev}(p,i,d)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cup d, \\ & \quad \mathrm{rev}(p,i,c)\Downarrow \leq_c \texttt{Object}, \mathrm{rev}(p,i,d)\Downarrow \leq_c \texttt{Object}, \\ & \quad \text{and } V_{P,i} = - \\ \mathrm{rev}(p,i,c)\Downarrow & \text{if } a\Downarrow = c\cap d, \mathrm{rev}(p,i,d)\Downarrow = \top, \text{ and } V_{P,i} = - \\ \mathrm{rev}(p,i,d)\Downarrow & \text{if } a\Downarrow = c\cap d, \mathrm{rev}(p,i,c)\Downarrow = \top, \text{ and } V_{P,i} = - \\ (\mathrm{rev}(p,i,c)\Downarrow \cup \mathrm{rev}(p,i,d)\Downarrow)\Downarrow & \text{if } a\Downarrow = c\cap d, \\ & \quad \mathrm{rev}(p,i,c)\Downarrow \leq_c \top, \mathrm{rev}(p,i,d)\Downarrow \leq_c \top, \text{ and } V_{P,i} = - \\ * & \text{if } a\Downarrow = * \\ \top & \text{otherwise} \end{cases}$$

D Typical representative

This algorithm produces a reduced type from a type specification. The difference between $\vdash_{TR}$ and $\vdash$ is that the latter produces a closed type.

$$\frac{\vdash_{TR} a :: A \qquad \vdash_{TR} b :: B}{\vdash_{TR} a\cup b :: A\cup B}\ \mathsf{TR}\cup \qquad \frac{\vdash_{TR} a :: A \qquad \vdash_{TR} b :: B}{\vdash_{TR} a\cap b :: A\cap B}\ \mathsf{TR}\cap$$

$$\frac{\vdash_{TR} a_1 :: A_1 \qquad \dots \qquad \vdash_{TR} a_{n_P} :: A_{n_P}}{\vdash_{TR} p(a_1,\dots,a_{n_P}) :: P(A_1,\dots,A_{n_P})}\ \mathsf{TRP}$$

$$\frac{\vdash_{TR} a :: A \qquad \vdash_{TR} b :: B}{\vdash_{TR} a\to b :: A\to B}\ \mathsf{TR}\to$$

$$\frac{}{\vdash_{TR} t :: T}\ \mathsf{TRConst} \qquad \frac{}{\vdash_{TR} x :: X}\ \mathsf{TRVar}$$

E Types

E.1 Reduced types

Reduced types are types with all reduction carried out.
ar ::= t (T is a user-defined type)
| `Object`
| `Behavior`
| $\top$
| $\bot$
| $p(ar_1, \ldots, ar_{n_P})$ (P is a user-defined parametric type)
| $ar_1 \cup ar_2$
| $ar_1 \cap ar_2$
| $\lambda x.a$ (only functions monotonic w.r.t $\leq_f$ are allowed)
| $ar \rightarrow br$ (shortcut for $\lambda x.(cond(\{x \leq ar, ar \leq \texttt{Object}\}); br))$

Closed reduced types (crtypes) are reduced types with no free variables.

Extended reduced types (ertypes) are like closed reduced types, but they can also contain the wildcard $*$ in any position a reduced type is allowed.

E.2 Argument types

Argument types are types that can be used in behavior argument specifications. Argument types are always reduced.
aa ::= t (T is a user-defined type)
| `Object`
| `Behavior`
| $p(aa_1, \ldots, aa_{n_P})$ (P is a user-defined parametric type)
| $aa_1 \cap aa_2$
| $aa \rightarrow ba$ (shortcut for $\lambda x.(cond(\{x \leq aa, aa \leq \texttt{Object}\}); ba))$

In argument types, arrow types are only allowed in positions covariant w.r.t. variance annotations.

Closed argument types (catypes) are argument types with no free variables.

Extended argument types (eatypes) are like closed argument types, but they can also contain the wildcard $*$ in any position an argument type is allowed.

E.3 Concrete types

Concrete types are types of objects. They are always reduced and closed and they can not be extended.
ac ::= t (T is a user-defined type)
| `Object`
| `Behavior`
| $p(ac_1, \ldots, ac_{n_P})$ (P is a user-defined parametric type)
| ⟨`ac-beh`⟩
⟨`ac-beh`⟩ ::= ⟨`ac-fun`⟩
| ⟨`ac-beh`⟩ ∩⟨`ac-fun`⟩
⟨`ac-fun`⟩ ::= $\lambda x.a$ (a is a type with no free variables except for x)
| $ac \rightarrow bc$ (shortcut for $\lambda x.(cond(\{x \leq ac, ac \leq \texttt{Object}\}); bc))$

In concrete types, behavior types ($\langle\texttt{ac-beh}\rangle$) and arrow types are only allowed in positions covariant w.r.t. variance annotations.

F Natural semantics of the toy language

$$(S, E \uplus \{\texttt{x} = \mathsf{o} : t\}, \texttt{x})\triangledown = (S, \mathsf{o} : t) \text{ where } \texttt{x} \text{ is a name}$$

$$(S, E, (\langle\texttt{exprs}\rangle))\triangledown = (S, E, \langle\texttt{exprs}\rangle)\triangledown$$

$$(S, E, \langle\texttt{class}\rangle(\texttt{x}, \langle\texttt{TS}\rangle))\triangledown = \begin{cases} (S, \mathsf{err}^\top : \top) & \text{if } \langle\texttt{x}\rangle \notin E \vee (\langle\texttt{x}\rangle = \mathsf{err}^\top : \top) \in E \\ (S, \mathsf{class}(t) : \texttt{Class}(t)) & \text{otherwise if } (\langle\texttt{x}\rangle = \mathsf{x} : x) \in E, \quad t = \mathrm{apply}(f, x)\Downarrow \end{cases}$$
$$\text{where } \langle\texttt{TS}\rangle \mapsto^+ S, \ \vdash f : S/C$$

$$(S, E, \texttt{fun (x)} \ \langle\texttt{expr}\rangle)\triangledown = (S, \mathsf{closure}(E, \texttt{x}, \langle\texttt{expr}\rangle) : \lambda x.s)$$
$$\text{where } E = \{\texttt{x}_i = \mathsf{o}_i : t_i\}, \quad \{\texttt{x}_i : t_i, \texttt{x} : x\} \triangleright \langle\texttt{expr}\rangle : s$$

$$(S, E, \langle\texttt{expr}\rangle_1; \langle\texttt{expr}\rangle_2)\triangledown = \begin{cases} (S_1, \mathsf{err}^\top : \top) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{err}^\top : \top) \\ (S_1, \mathsf{err}^\perp : \perp) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{err}^\perp : \perp) \\ (S_2, \mathsf{o}_2 : t_2) & \text{otherwise, where } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{o}_1 : t_1), \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\triangledown = (S_2, \mathsf{o}_2 : t_2) \end{cases}$$

$$(S, E, \texttt{let x} = \langle\texttt{expr}\rangle_1 \ \texttt{in} \ \langle\texttt{expr}\rangle_2)\triangledown = \begin{cases} (S_1, \mathsf{err}^\top : \top) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{err}^\top : \top) \\ (S_1, \mathsf{err}^\perp : \perp) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{err}^\perp : \perp) \\ (S_2, \mathsf{o}_2 : t_2) & \text{otherwise, where } (S, E, \langle\texttt{expr}\rangle_1)\triangledown = (S_1, \mathsf{o}_1 : t_1), \\ & (S_1, E \uplus \{\texttt{x} = \mathsf{o}_1 : t_1\}, \langle\texttt{expr}\rangle_2)\triangledown = (S_2, \mathsf{o}_2 : t_2) \end{cases}$$

$$(S, E, \langle\texttt{expr}\rangle_1(\langle\texttt{expr}\rangle_2))\bigtriangledown = \begin{cases} (S_1, \mathsf{err}^\top : \top) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{err}^\top : \top) \\ (S_1, \mathsf{err}^\perp : \perp) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{err}^\perp : \perp) \\ (S_2, \mathsf{err}^\top : \top) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_1, \mathsf{err}^\top : \top) \\ (S_2, \mathsf{err}^\perp : \perp) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_1, \mathsf{err}^\perp : \perp) \\ (S_3, \mathsf{o}_3 : t_3) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{b} : t_1) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_2, \mathsf{o}_2 : t_2) \\ & dispatch(b, t_2) = i \\ & b_i = \langle\texttt{TS}\rangle : \texttt{fun (x)}\ \langle\texttt{expr}\rangle_3) \\ & (S_2, E \uplus \{x = \mathsf{o}_2 : t_2\}, \langle\texttt{expr}\rangle_3)\bigtriangledown = (S_3, o_3 : t_3) \\ (S_3, \mathsf{o}_3 : t_3) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{b} : t) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_2, \mathsf{o}_2 : t_2) \\ & dispatch(b, t_2) = i \\ & b_i = \langle\texttt{TS}\rangle : \texttt{primitive}_j \\ & (S_2, \mathsf{o}_2 : t_2)\bigtriangledown_{prim_j} = (S_3, \mathsf{o}_3 : t_3) \\ (S_3, \mathsf{o}_3 : t_3) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{closure}(E_c, \texttt{x}, \langle\texttt{expr}\rangle_3) : f) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_2, \mathsf{o}_2 : t_2) \\ & (S_2, E \uplus E_c \uplus \{x = \mathsf{o}_2 : t_2\}, \langle\texttt{expr}\rangle)\bigtriangledown = (S_3, \mathsf{o}_3 : t_3) \\ (S_2, \mathsf{err}^\top : \top) & \text{otherwise, where } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & (S_1, E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S_2, \mathsf{o}_2 : t_2) \end{cases}$$

$$(S, E, \langle\langle\texttt{expr}\rangle_1, \ldots, \langle\texttt{expr}\rangle_n\rangle)\bigtriangledown = \begin{cases} (S_{i+1}, \mathsf{err}^\top : \top) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & \vdots \\ & (S_i, E, \langle\texttt{expr}\rangle_{i+1})\bigtriangledown = (S_{i+1}, \mathsf{err}^\top : \top) \\ & \text{and } \mathsf{o}_i \neq \mathsf{err}^\top,\ \mathsf{o}_i \neq \mathsf{err}^\perp \\ (S_{i+1}, \mathsf{err}^\perp : \perp) & \text{if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & \vdots \\ & (S_i, E, \langle\texttt{expr}\rangle_{i+1})\bigtriangledown = (S_{i+1}, \mathsf{err}^\perp : \perp) \\ & \text{and } \mathsf{o}_i \neq \mathsf{err}^\top,\ \mathsf{o}_i \neq \mathsf{err}^\perp \\ (S_n, \langle\mathsf{o}_1, \ldots, \mathsf{o}_n\rangle) & \text{otherwise, if } (S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S_1, \mathsf{o}_1 : t_1) \\ & \vdots \\ & (S_{n-1}, E, \langle\texttt{expr}\rangle_n)\bigtriangledown = (S_n, \mathsf{o}_n : t_n) \\ & \text{and } \mathsf{o}_i \neq \mathsf{err}^\top,\ \mathsf{o}_i \neq \mathsf{err}^\perp \end{cases}$$

G Proofs

Theorem G.1. *Behavior consistency:* If behavior b is consistent, $b : tb = \cap_{i=1,ldots,n} s_i \in \Theta_0$, and t is a concrete type such that $\text{apply}(tb, t)\Downarrow \leq_f$ `Object`, then

1. $dispatch(b, t) \neq 0$
2. If $dispatch(b, t) = i$ then

$$\text{apply}(s_i, t)\Downarrow =_f \text{apply}(tb, t)\Downarrow$$

3. If $dispatch(b, t) = i$, function $\langle\texttt{function}\rangle_i$ is a part of this association, and $\Theta_0 \triangleright \langle\texttt{function}\rangle_i : f_i$, then

$$\text{apply}(f_i, t)\Downarrow \leq_f \text{apply}(tb, t)\Downarrow$$

□

Proof:

1. We will first prove that the dispatch in consistent and then that it is unambiguous. In order to prove consistency, it is sufficient to show that under the conditions of the theorem

$$\exists i: \quad \text{apply}(s_i, t)\Downarrow \leq_f \texttt{Object}$$

where $\langle\texttt{TS}\rangle_i$ are b's specifications, $\langle\texttt{TS}\rangle_i \mapsto^+ S_i$, and $\vdash s_i :: S_i$. Since $tb = \cap_{i=1,ldots,n} s_i$ and $\text{apply}(tb, t)\Downarrow \leq_f$ `Object` according to the conditions of the theorem, consistency trivially follows from the rules for $\leq_f$ and $\Downarrow$.
Now we have to show consistency. Let i' be a number such that $\text{apply}(s_{i'}, t)\Downarrow \leq_f$ `Object` (we already know that such i' exists). Then to prove consistency it is sufficient to show that under the conditions of the theorem

$$\forall i: \quad \text{apply}(s_i, t)\Downarrow \leq_f \texttt{Object} \Rightarrow \langle\texttt{TS}\rangle_{i'} \leq_{\text{cov}} \langle\texttt{TS}\rangle_i$$

We will prove this statement by contradiction. Let us assume that there is $i'' \neq i'$ such that

$$\text{apply}(s_{i''}, t)\Downarrow \leq_f \texttt{Object} \wedge$$
$$(\neg\exists i''': \quad \langle\texttt{TS}\rangle_{i'''} \leq_{\text{cov}} \langle\texttt{TS}\rangle_{i'} \wedge \langle\texttt{TS}\rangle_{i'''} \leq_{\text{cov}} \langle\texttt{TS}\rangle_{i''}) \wedge \text{apply}(s_{i'''}, t)\Downarrow \leq_f \texttt{Object}$$

According to the Corollary 5.1, this means that there exist such concrete types $\vec{t''}$ and $\vec{t'}$ that $t \leq_f [\vec{t'}/\vec{x}']arg_{i'}$ and $t \leq_f [\vec{t''}/\vec{x}'']arg_{i''}$, where $\vdash_{TR} arg_i :: \langle\texttt{TS}\rangle_i$. Thus (according to the Theorem 7.1)

$$\exists \hat{t} \in [\vec{*}/\vec{x}']arg_{i'} \cap_? [\vec{*}/\vec{x}'']arg_{i''}, \vec{ac}: \quad t \leq_f [\vec{ac}/\vec{*}]\hat{t}$$

Since according to the conditions of the theorem b is consistent, from the definition of consistency we have

$$\exists! i''': \quad \langle\texttt{TS}\rangle_{i'''} \leq_{\text{cov}} \langle\texttt{TS}\rangle_1 \wedge \langle\texttt{TS}\rangle_{i'''} \leq_{\text{cov}} \langle\texttt{TS}\rangle_2 \wedge \text{apply}(s_{i'''}, [\vec{y}/\vec{*}]\hat{t}) \leq_c \texttt{Object}$$

where $\vec{y}$ are fresh variables. It is easy to show that

$$\forall \vec{ac}[\vec{y}/\vec{*}]\hat{t} \leq_f \texttt{Object} \Rightarrow [\vec{ac}/\vec{*}]\hat{t} \leq_f \texttt{Object}$$

where $\vec{y}$ are fresh (anything is better than an unbound variable for subtyping derivation). Thus, we have that

$$\text{apply}(s_{i'''}, [\vec{ac}/\vec{*}]\hat{t})\Downarrow \leq_c \texttt{Object}$$
$$t \leq_f [\vec{ac}/\vec{*}]\hat{t}$$

From this, Theorem 4.1, and Corollary 5.1 we have

$$\text{apply}(s_{i'''}, t)\Downarrow \leq_f \texttt{Object}$$

Thus, we have a contradiction as the association number i''' is more specific than both i' and i''.

2. Let i' be the number of the most specific association. Then

$$\forall i \neq i'\colon \quad \neg(\text{apply}(s_i, t)\Downarrow \leq_f \texttt{Object}) \vee \langle\texttt{TS}\rangle_{i'} \leq_{\text{cov}} \langle\texttt{TS}\rangle_i$$

If

$$\neg(\text{apply}(s_i, t)\Downarrow \leq_f \texttt{Object})$$

then

$$\text{apply}(s_i, t)\Downarrow = \top$$

and everything is fine as

$$\text{apply}(s_{i'}, t)\Downarrow \leq_f \texttt{Object} \leq_f \top$$

If

$$\text{apply}(s_i, t)\Downarrow \leq_f \texttt{Object} \wedge \langle\texttt{TS}\rangle_{i'} \leq_{\text{cov}} \langle\texttt{TS}\rangle_i$$

then (by the definition of $\leq_{\text{cov}}$) $r_{i'} \leq_c r_i$ and thus $r_{i'} \leq_f r_i$, where $\vdash_{TR} a_j \rightarrow r_j :: \langle\texttt{TS}\rangle_j$. From the Corollary 5.1 it follows that

$$\forall t\colon \quad \text{apply}(s_i, t) \leq_f \texttt{Object} \exists \vec{u}\colon \quad \text{apply}(s_i, t) = [\vec{u}/\vec{x}]r_i$$

Therefore,

$$\exists \vec{u}\colon \quad \text{apply}(s_i, t) = [\vec{u}/\vec{x}]r_i$$
$$\exists \vec{u'}\colon \quad \text{apply}(s_{i'}, t) = [\vec{u'}/\vec{x'}]r_{i'}$$

It is easy to show that

$$\forall \vec{u}, \vec{u'}\colon \quad a \leq_f b \Rightarrow [\vec{u}/\vec{x}]a \leq_f [\vec{u'}/\vec{b'}]b$$

as unbound variables are treated by $\leq_f$ just like a type whose only property is that it is less than `Object`, and thus substituting any type for them will not affect the derivation of subtyping. From this we have

$$\text{apply}(s_{i'}, t)\Downarrow \leq_f \text{ apply}(s_i, t)\Downarrow$$

and therefore we have

$$\text{apply}(s_{i'}, t)\Downarrow \leq_f \text{ apply}(s_i, t)\Downarrow$$

for all $i \neq i'$. Then

$$\text{apply}(bt, t)\Downarrow = \text{apply}(\cap_{i \neq i'} s_i \cap s_{i'}, t)\Downarrow = \cap_{i \neq i'} \text{apply}(s_i, t) \cap \text{apply}(s_{i'}, t)\Downarrow = \text{apply}(s_{i'}, t)\Downarrow$$

3. In order to prove this statement it is sufficient to prove that under the conditions of the theorem

$$\text{apply}(f_i, t)\Downarrow \leq_f \text{ apply}(s_i, t)\Downarrow$$

since we have already shown that

$$\text{apply}(s_i, t)\Downarrow =_f \text{ apply}(tb, t)\Downarrow$$

Under the conditions of the theorem the association i is consistent and therefore (according to the definition of consistency)

$$f_i \leq_c s_i$$

and thus $f_i \leq_f s_i$. Now according to the rule $\lambda^{<>}$ we have that

$$\text{apply}(f_i, t)\Downarrow \leq_f \text{ apply}(s_i, t)\Downarrow$$

and thus we obtained the proof of the statement of the theorem.

Theorem G.2. *Subject reduction:* If

1. The user type graph $\mathcal{G}$ is consistent
2. All behaviors are consistent
3. All primitive function associations are consistent
4. Store S is consistent
5. All free object variables $\mathtt{x}_i$ in $\langle\mathtt{expr}\rangle$ are present in E
6. $E = \{\mathtt{x}_i = \mathtt{o}_i : t_i\}$, $\Theta = \{\mathtt{x}_i : t_i\}$, $\Theta \triangleright \langle\mathtt{expr}\rangle : t_e$, and $t_e\Downarrow \leq_f$ `Object`
7. $(S, E, \langle\mathtt{expr}\rangle)\bigtriangledown = (S', \mathtt{o} : q)$

then

1. S' is consistent
2. $q \leq_f t_e\Downarrow$

□

The consequence of this theorem is that the reduction of type-correct terms does not produce run-time type errors ($\mathsf{err}^\top$) or inconsistent store. Note that other run-time errors ($\mathsf{err}^\bot$) can only be produced by primitive functions, if they are so defined.

Detailed sketch of the proof: The proof is by structural induction on $\bigtriangledown$. The most involved case is the one dealing with application ($\langle\mathtt{expr}\rangle_1(\langle\mathtt{expr}\rangle_2)$) when neither of the expressions is reduced to $\mathsf{err}^\bot$. There are four main cases:

1. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown \neq (S'', \mathsf{b} : bt) \quad \wedge \quad (S, E, \langle\mathtt{expr}\rangle_1\bigtriangledown) \neq (S'', \mathsf{closure}(E_c, \mathtt{x}, \langle\mathtt{expr}\rangle) : c)$ (bad)
2. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{b} : bt)$, $(S'', E, \langle\mathtt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$, and $dispatch(b, t) = 0$ (bad)
3. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{b} : bt)$ $(S'', E, \langle\mathtt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$, and $dispatch(b, t) = i$, $\mathtt{fun}\ (x)\ \langle\mathtt{expr}\rangle$ is a part of the i-th association of b (good)
4. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{b} : bt)$, $(S'', E, \langle\mathtt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$, and $dispatch(b, t) = i$, $\mathtt{primitive}_i$ is a part of the i-th association of b (good)
5. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{closure}(E_c, \mathtt{x}, \langle\mathtt{expr}\rangle) : c)$, and $(S'', E, \langle\mathtt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$ (good)

We have to show that bad cases never happen, while good cases do not produce $\mathsf{err}^\top$.

1. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown \neq (S'', \mathsf{b} : bt) \quad \wedge \quad (S, E, \langle\mathtt{expr}\rangle_1\bigtriangledown) \neq (S'', \mathsf{closure}(E_c, \mathtt{x}, \langle\mathtt{expr}\rangle) : c)$ (bad) Assume $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{o} : t_o)$ Then $t_o \not\leq_f \mathtt{Behavior}$ since behaviors closures are the only objects (except for $\mathsf{err}^\bot$ which we do not consider here) that have subtypes of $\mathtt{Behavior}$ as types.

   $$\Theta \rhd \langle\mathtt{expr}\rangle_1(\langle\mathtt{expr}\rangle_2) : t_e,\ t_e\Downarrow \leq_f \mathtt{Object}$$

   (assumption of the theorem) and

   $$\Theta \rhd \langle\mathtt{expr}\rangle_1 : t_1,\ t_1\Downarrow \geq_f t_o$$

   (induction assumption). According to Appl,

   $$t_e = \mathrm{apply}(t_1, t_2)\Downarrow$$

   But according to the rules for $\Downarrow$ and $\leq_f$

   $$\mathrm{apply}(t_1, t_2)\Downarrow \leq_f \mathtt{Object} \Rightarrow t_1\Downarrow \leq_f \mathtt{Behavior}$$

   Thus we have

   $$\mathtt{Behavior} \not\geq_f\ t_o \leq_f t_1\Downarrow \leq_f \mathtt{Behavior}$$

   which is a contradiction. Thus this case can never happen.
2. $(S, E, \langle\mathtt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{b} : bt)$, $(S'', E, \langle\mathtt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$, and $dispatch(b, t) = 0$ We assume $\Theta \rhd \langle\mathtt{expr}\rangle_2 : t_2$ Then

   $$\mathrm{apply}(bt, t_2)\Downarrow \not\leq_f \mathtt{Object}$$

   according to the Theorem 7.2 and the definition of *dispatch*. According to the condition of this theorem and the rule Appl

   $$\Theta \rhd \langle\mathtt{expr}\rangle_1(\langle\mathtt{expr}\rangle_2) : \mathrm{apply}(t_1, t_2),\ \mathrm{apply}(t_1, t_2)\Downarrow \leq_f \mathtt{Object}$$

where

$$\Theta \triangleright \langle \texttt{expr} \rangle_1 : t_1$$

and $bt \leq_f t_1 \Downarrow$ by induction assumption. Then by the monotonicity of apply w.r.t $\leq_f$

$$\text{apply}(bt, t_2)\Downarrow \leq_f \text{apply}(t_1, t_2)\Downarrow$$

and we have

$$\texttt{Object} \not\geq_f \text{apply}(bt, t_2)\Downarrow \leq_f \text{apply}(t_1, t_2)\Downarrow \leq_f \texttt{Object}$$

which is a contradiction. Thus this situation can never happen.

3. $(S, E, \langle \texttt{expr} \rangle_1)\bigtriangledown = (S'', \texttt{b} : bt)$ $(S'', E, \langle \texttt{expr} \rangle_2)\bigtriangledown = (S''', \texttt{p} : t)$, and $dispatch(b, t) = i$, $\texttt{fun}\ (x)\ \langle \texttt{expr} \rangle$ is a part of the i-th association of b. Let us put

$$\Theta \triangleright \langle \texttt{expr} \rangle_1 : t_1$$
$$\Theta \triangleright \langle \texttt{expr} \rangle_2 : t_2$$
$$\Theta \triangleright (\texttt{fun}\ (x)\ \langle \texttt{expr} \rangle) : f$$

Then we have

$$t_e\Downarrow = \text{apply}(t_1, t_2)\Downarrow \text{ by the rule } \mathsf{Appl}$$
$$bt \leq_f t_1\Downarrow \text{ by induction hypothesis}$$
$$t \leq_f t_2\Downarrow \text{ by induction hypothesis}$$

Therefore (by the rule $\lambda^{<>}$ ($\leq_f$), Corollary 5.1, and transitivity of $\leq_f$)

$$\text{apply}(bt, t)\Downarrow \leq_f t_e\Downarrow$$

By the Theorem 7.2 and transitivity

$$\text{apply}(f, t)\Downarrow \leq_f t_e\Downarrow$$

By the rule Fun we have

$$f = \lambda x.e, \text{ where } \Theta_0, \texttt{x} : x \triangleright \langle \texttt{expr} \rangle : e$$

By the rule for $\Downarrow$ we have

$$\text{apply}(f, t)\Downarrow = [t/x]e\Downarrow$$

and by the rule for $\bigtriangledown$ we have

$$(S, E, \langle \texttt{expr} \rangle_1(\langle \texttt{expr} \rangle_2))\bigtriangledown = (S''', E \uplus \{\texttt{x} = \texttt{p} : t\}, \langle \texttt{expr} \rangle)\bigtriangledown$$

We now have to show that

$$\begin{aligned}
&q\Downarrow \leq_f [t/x]e\Downarrow \text{ and } S' \text{ is consistent,}\\
&\quad \text{where } (S''', E', \langle\texttt{expr}\rangle)\bigtriangledown = (S', \texttt{o} : q),\\
&\quad E' = E\uplus\{\texttt{x} = \texttt{p} : t\},\\
&\quad \Theta' \triangleright \langle\texttt{expr}\rangle : e,\\
&\quad \Theta' = \Theta\cup\{\texttt{x} : x\},\\
&\quad S''' \text{ is consistent,}\\
&\quad \text{all free object variables in } \langle\texttt{expr}\rangle \text{ are present in } E',\\
&\quad [t/x]e\Downarrow \leq_f \texttt{Object} \text{ (by the Theorem 7.2)}
\end{aligned}$$

but this is the same as

$$\begin{aligned}
&q\Downarrow \leq_f e'\Downarrow \text{ and } S' \text{ is consistent,}\\
&\quad \text{where } (S''', E', \langle\texttt{expr}\rangle)\bigtriangledown = (S', \texttt{o} : q),\\
&\quad E' = E\uplus\{\texttt{x} = \texttt{p} : t\},\\
&\quad \Theta' \triangleright \langle\texttt{expr}\rangle : e',\\
&\quad \Theta' = \Theta\cup\{\texttt{x} : t\},\\
&\quad S''' \text{ is consistent,}\\
&\quad \text{all free object variables in } \langle\texttt{expr}\rangle \text{ are present in } E',\\
&\quad e'\Downarrow \leq_f \texttt{Object},\\
&\quad e' = [t/x]e
\end{aligned}$$

which is the induction hypothesis. Thus, this case is proven.

4. $(S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{b} : bt)$, $(S'', E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$, and $dispatch(b, t) = i$, $\texttt{primitive}_i$ is a part of the i-th association of b. In this case the statement of the theorem directly follows from the definition of primitive function association consistency and from the induction hypothesis. Note that this is the only case when store consistency is an issue as the store can only be directly manipulated upon by primitive functions.
5. $(S, E, \langle\texttt{expr}\rangle_1)\bigtriangledown = (S'', \mathsf{closure}(E_c, \texttt{x}, \langle\texttt{expr}\rangle) : c)$, and $(S'', E, \langle\texttt{expr}\rangle_2)\bigtriangledown = (S''', \mathsf{p} : t)$. This is analogous to the case 4 but is much simpler since dispatch is not involved.

Other (non-application) cases are significantly simpler.