Explorations of Type Systems

Semester Project Report

Samuel Grütter Supervisor: Nada Amin
EPFL
samuel.gruetter@epfl.ch

Contents

1 Introduction **1**

2 A tour exploring type systems **1**
2.1 A simple structurally typed language 1
2.2 μ-Types 5
2.3 EDOT: DOT as existential types 7
2.4 Trying to generalize EDOT to DOT 10
2.5 Infinite paths 10

3 Side topics **11**
3.1 Types and mathematical sets 11
3.2 Non-collapsed bounds in concrete types . . 12
3.3 Bugs detected 12

4 Conclusion **12**

1. Introduction

The original intention for the semester project was to explore structural type systems. As a first step, I implemented a simple structurally typed language in PLT redex [3], described in section 2.1.

It had the serious limitation that no recursive types were allowed, so I started investigating μ-Types (section 2.2).

At the same time, I also started studying the paper on Dependant Object Types [2], and realized that the simple structurally typed language augmented with μ-Types would still be less powerful than DOT, so I became interested in DOT. Comparing DOT to what I had seen before in type systems, I detected similarities to existential types, and explored these in the "EDOT" project (section 2.3). This lead to a quite different type system which models a subset of DOT, and can prove some subtype relationships that DOT as described in [2] cannot prove (but the current version can).

Then I tried to generalize the ideas from this "EDOT" project to DOT, but had to see that this is not possible (section 2.4).

But trying to do so, I started understanding how challenging path types which can become arbitrarily long are, and wanted to figure out whether DOT can deal with all these situations. This work is described in section 2.5, and includes an example on which the current DOT implementation (and the earlier ones, too) runs into an infinite loop. A solution to this problem is proposed, but without any proof.

Besides the topics encountered in the journey described above, I also treated some other topics: The relationship between types and mathematical sets in section 3.1, the purpose of having distinct lower and upper bounds for type members of concrete types (section 3.2), and also detected two bugs in the DOT implementation, described in section 3.3.

I implemented most of the studied systems in PLT redex [3]. The source code can be found at

`https://github.com/samuelgruetter/type-systems-spring13`

2. A tour exploring type systems

2.1 A simple structurally typed language

The first part of the semester project was to design a small and simple structurally typed language, named SimpleV2, and to implement it in PLT Redex [3].

2.1.1 Syntax

The syntax is presented in Figure 1. We distinguish between expressions, which return a value, and statements, which return nothing (not even **void**). An expression returning **void** can be turned into a statement by preceding it with the **ign** keyword. Value declarations and type declarations are also allowed wherever a statement is allowed. Note that value declarations in SimpleV2 rather correspond to initializations in DOT than to value declarations. Value declarations and type declarations can never be overridden.

There are the primitive types **Int** , **Bool** , **Str** , and literals for them. **Void** is also a type, but is not considered as primitive type, and the only value of type **Void** is **void** .

The non-primitive types appear in two flavors: Simplified types S, which cannot be reduced to something simpler, and type expressions T, which need not necessarily be in fully simplified form.

Interface types S_i and T_i specify a set of value members that an object must have in order to belong to the type. By giving a sequence of value declarations and type declarations, one can construct an object, whose type is an interface

type. Note that type declarations are allowed in the object construction, but can only be used inside the object, and do not appear in interface types.

Compared to DOT, type declarations are much less powerful: They only are type aliases, i.e. a way to avoid rewriting the same type over and over again, but do not add any more power to the type system. Unfortunately, this was only detected once the language was implemented in PLT redex and compared to DOT.

Function types $S_i \to S_i$ and $T_i \to T_i$ are the type of anonymous functions $(x : T) \Rightarrow e$, which always take one argument and return one value. Argument type and return type both can be **Void** .

Var types (**var** S) and (**var** T) are the type of reference cells (**cell** e), which store mutable data. Cells have a get method to retrieve the value stored in the cell, and a set method to assign a value. This is a built-in feature of the language.

There are also intersection types $T_1 \wedge T_2$, which specify that objects must belong to both T_1 and T_2. An intersection of two types of different kinds (for instance, a primitive type and a function type, or a var type and an interface type) is not rejected by the syntax, but will be rejected by the typechecker.

Moreover, identifiers X are also allowed in type expressions, but they should refer to a type alias in scope, because otherwise the type expression will not typecheck.

Note that we use the same set of identifiers for value declarations, type declarations, and also for the argument of anonymous functions. We use the symbols x, y, X, Y to denote such identifiers. For better readability, we will use X and Y for identifiers referring to a type declaration, and x and y for identifiers referring to a value declaration or the argument of an anonymous function, but this is only a convention and is not specified by the grammar.

The expressions not already discussed above with the types are function applications $(e_1\ e_2)$, where the type of e_1 should be a function type in order to pass typechecking, block expressions, which consist of some statements followed by an expression (which can be **void** to get a block with no return value), if-then-else statements, which return a (possibly **void**) value, value member selections $e.x$, and some binary operations on integers. To make the language more interesting, we would need more binary operators for integers, and also logical operators for booleans, but these were omitted to keep the description short, and because it is straightforward how to add them.

Typing environments Γ contain the types of all values in the current scope, but also the types that were assigned to identifiers in type declarations. For type declarations and value declarations, the same set of identifiers is used, so it is not allowed to have a value and a type alias with the same name. It is not allowed to redefine a value declaration or a type declaration which is already in scope, no matter whether it was declared in the same scope or in an outer scope.

2.1.2 Type simplification

Before type expressions T are used for typechecking or put into a typing environment Γ, they are simplified to S following the rules in Figure 2. $T \mapsto S$ means that the the simplified form of T is S. It is unique up to the ordering of the members in interface types.

The **intersect** function in the ($\mapsto$-$\wedge$) rule takes two simplified types S_1 and S_2, and returns a simplified type S, which is equivalent to $S_1 \wedge S_2$, or **false** if S_1 and S_2 are a different kind of type. The exact implementation is given in the Redex model.

2.1.3 Type assignment

The type assignment rules are presented in Figure 2. Most of them are similar to other languages such as Scala, and are not further discussed here. Still, some points are worth being considered:

In the IF rule, the **union** function appears, even though the type system does not have union types. The **union** function checks if one of its arguments is a subtype of the other, and if so, it returns the supertype, and otherwise, it fails, so not all **if** statements will typecheck.

Typechecking of blocks is done recursively, using the BLOCK_1, BLOCK_2, and BLOCK_3 rules as steps and the BLOCK_0 rule as base case. Typechecking of object constructions works similarly, except that there are no **ign** allowed in object constructions.

Notice that the BLOCK rules and OC rules are such that identifiers can only refer to values and types declared earlier, so no cyclic references are possible. This severely limits the power of the language, because no recursive types are possible, and types where one has a reference to the second and the second has a reference to the first are not possible either.

Type assignment rules always return a simplified type S, and also work with simplified types. If there are types which come from the source code of the program, they are first simplified (see rules ANON, BLOCK_2 and OC_2).

In the rules, adding a mapping to Γ is written as Γ, m. This operation has the following semantics: If there is already a mapping in Γ with the same identifier as m, the operation fails, and the rule in which it appears cannot be applied. Otherwise, $\Gamma, m = \Gamma \cup \{m\}$.

2.1.4 Subtyping

The subtyping rules are presented in Figure 1. **Void** and primitive types are subtypes of themselves, function argument types are contravariant, and function return types are covariant, and var types are nonvariant.

Subtyping of interface types is defined by recursion on the length of the right-hand side type: Each time the ($<:$-INTF-STEP) rules is applied, the length of the right-hand side

Syntax

$s ::=$		statement
	(**ign** e)	ignore return value of expression
	d	type or value declaration
$d ::=$		declaration
	d_v	value declaration
	d_t	type declaration
$d_v ::= (\textbf{val}\ x = e)$		value declaration
$d_t ::= (\textbf{type}\ X = T)$		type declaration (alias)
$P ::=$		primitive type
	Int	
	Bool	
	Str	
$S ::=$		simplified type
	Void	the type of **void**
	P	primitive type
	S_i	simplified interface type
	$S \to S$	simplified function type
	(**var** S)	simplified var type
$T ::=$		type expression
	Void	the type of **void**
	P	primitive type
	T_i	interface type
	$T \to T$	function type
	(**var** T)	type of reference cells
	$T \wedge T$	intersection type
	X	type alias referring to type

$S_i ::=$		simplified interface type
	$\{\overline{(\textbf{val}\ x : S)}\}$	
$T_i ::=$		interface type
	$\{\overline{(\textbf{val}\ x : T)}\}$	
$e ::=$		expression
	$(e\ e)$	function application
	c	object construction
	x, y, X, Y	identifiers
	$\{\overline{s}\ e\}$	block expression
	$(x : T) \Rightarrow e$	anonymous function
	(**if** $e\ e\ e$)	if-then-else returning value
	e_{bin}	binary operation
	$e.x$	value member selection
	(**cell** e)	cell storing mutable data
	e_{lit}	literal
$c ::= \overline{d}$		object construction
$e_{bin} ::=$		binary expression
	$e < e \mid e + e$	
$e_{lit} ::=$		literal
	$number \mid \textbf{true} \mid \textbf{false} \mid \textbf{void} \mid string$	
$\Gamma ::= \overline{m}$		typing environment
$m ::=$		"mapping"
	$x : S$	maps var identifier to its type
	$X : S$	maps type identifier to a type

Subtyping $\boxed{S_1 <: S_2}$

$$\frac{}{\textbf{Void} <: \textbf{Void}}\ (<:\text{-VOID}) \qquad \frac{S_{21} <: S_{11}\ ,\ S_{12} <: S_{22}}{S_{11} \to S_{12} <: S_{21} \to S_{22}}\ (<:\text{-FUNC})$$

$$\frac{}{P <: P}\ (<:\text{-P})$$

$$\frac{}{S_i <: \{\ \}}\ (<:\text{-INTF-BASE}) \qquad \frac{S_1 <: S_2\ ,\ S_2 <: S_1}{(\textbf{var}\ S_1) <: (\textbf{var}\ S_2)}\ (<:\text{-VAR})$$

$$\frac{S_b <: S_a}{\{\overline{(\textbf{val}\ x_{b1} : S_{b1})}\ (\textbf{val}\ x : S_b)\ \overline{(\textbf{val}\ x_{b2} : S_{b2})}\} <: \{\overline{(\textbf{val}\ x_{a1} : S_{a1})}\}}{\{\overline{(\textbf{val}\ x_{b1} : S_{b1})}\ (\textbf{val}\ x : S_b)\ \overline{(\textbf{val}\ x_{b2} : S_{b2})}\} <: \{(\textbf{val}\ x : S_a)\ \overline{(\textbf{val}\ x_{a1} : S_{a1})}\}}\ (<:\text{-INTF-STEP})$$

Figure 1. The SimpleV2 Calculus: Syntax and Subtyping

Type simplification $\boxed{\Gamma \vdash T \mapsto S}$

$$\frac{\Gamma \vdash \overline{T \mapsto S}}{\Gamma \vdash \{\overline{(\textbf{val}\ x : T)}\} \mapsto \{\overline{(\textbf{val}\ x : S)}\}}\ (\mapsto\text{-INTF}) \qquad \frac{(X : S) \in \Gamma}{\Gamma \vdash X \mapsto S}\ (\mapsto\text{-LOOKUP})$$

$$\frac{\Gamma \vdash T_1 \mapsto S_1\ ,\ T_2 \mapsto S_2}{\Gamma \vdash (T_1 \to T_2) \mapsto (S_1 \to S_2)}\ (\mapsto\text{-FUNC}) \qquad \frac{\Gamma \vdash T_1 \mapsto S_1\ ,\ T_2 \mapsto S_2}{\Gamma \vdash (T_1 \wedge T_2) \mapsto \textbf{intersect}(S_1, S_2)}\ (\mapsto\text{-}\wedge)$$

$$\frac{\Gamma \vdash T \mapsto S}{\Gamma \vdash (\textbf{var}\ T) \mapsto (\textbf{var}\ S)}\ (\mapsto\text{-VAR})$$

Type assignment $\boxed{\Gamma \vdash e : S}$

$$\frac{S_a <: S'_a \qquad \Gamma \vdash e_f : S'_a \to S_r\ ,\ e_a : S_a}{\Gamma \vdash (e_f\ e_a) : S_r}\ (\text{APPL}) \qquad \frac{\Gamma \vdash e_1 : \textbf{Void} \qquad \Gamma \vdash \{\overline{s}\ e_2\} : S_2}{\Gamma \vdash \{(\textbf{ign}\ e_1)\ \overline{s}\ e_2\} : S_2}\ (\text{BLOCK}_3)$$

$$\frac{\Gamma \vdash T_1 \mapsto S_1 \qquad \Gamma, (x : S_1) \vdash e : S_2}{\Gamma \vdash (x : T_1) \Rightarrow e : S_1 \to S_2}\ (\text{ANON}) \qquad \frac{}{\Gamma \vdash (\ ) : \{\ \}}\ (\text{OC}_0)$$

$$\frac{(x : S) \in \Gamma}{\Gamma \vdash x : S}\ (\text{LOOKUP}) \qquad \frac{\Gamma \vdash e : S \qquad \Gamma, (\textbf{val}\ x : S) \vdash (\overline{d}) : \{\overline{(\textbf{val}\ y : S')}\}}{\Gamma \vdash ((\textbf{val}\ x = e)\ \overline{d}) : \{(\textbf{val}\ x : S)\ \overline{(\textbf{val}\ y : S')}\}}\ (\text{OC}_1)$$

$$\frac{}{\Gamma \vdash number : \textbf{Int}}\ (\text{INT-LIT}) \qquad \frac{\Gamma \vdash T \mapsto S \qquad \Gamma, (\textbf{type}\ X : S) \vdash (\overline{d}) : \{\overline{(\textbf{val}\ y : S')}\}}{\Gamma \vdash ((\textbf{type}\ X = T)\ \overline{d}) : \{\overline{(\textbf{val}\ y : S')}\}}\ (\text{OC}_2)$$

$$\frac{}{\Gamma \vdash string : \textbf{Str}}\ (\text{STR-LIT}) \qquad \frac{\Gamma \vdash e : \{\overline{(\textbf{val}\ y_1 : S_1)}\ (\textbf{val}\ x : S)\ \overline{(\textbf{val}\ y_2 : S_2)}\}}{\Gamma \vdash e.x : S}\ (\text{SEL})$$

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{Bool}\ ,\ \textbf{false} : \textbf{Bool}}\ (\text{BOOL-LIT}) \qquad \frac{\Gamma \vdash e : S}{\Gamma \vdash (\textbf{cell}\ e) : (\textbf{var}\ S)}\ (\text{CELL})$$

$$\frac{}{\Gamma \vdash \textbf{void} : \textbf{Void}}\ (\text{VOID-LIT}) \qquad \frac{\Gamma \vdash e : (\textbf{var}\ S)}{\Gamma \vdash e.\textbf{get} : \textbf{Void} \to S}\ (\text{CELL-GET})$$

$$\frac{\Gamma \vdash e_1 : \textbf{Bool}\ ,\ e_2 : S_2\ ,\ e_3 : S_3}{\Gamma \vdash (\textbf{if}\ e_1\ e_2\ e_3) : \textbf{union}(S_2, S_3)}\ (\text{IF}) \qquad \frac{\Gamma \vdash e : (\textbf{var}\ S)}{\Gamma \vdash e.\textbf{set} : S \to \textbf{Void}}\ (\text{CELL-SET})$$

$$\frac{\Gamma \vdash e : S}{\Gamma \vdash \{e\} : S}\ (\text{BLOCK}_0) \qquad \frac{\Gamma \vdash e_1 : \textbf{Int}, e_2 : \textbf{Int}}{\Gamma \vdash e_1 + e_2 : \textbf{Int}}\ (\text{INT-}+)$$

$$\frac{\Gamma \vdash e_1 : S_1 \qquad \Gamma, (x : S_1) \vdash \{\overline{s}\ e_2\} : S_2}{\Gamma \vdash \{(\textbf{val}\ x = e_1)\ \overline{s}\ e_2\} : S_2}\ (\text{BLOCK}_1) \qquad \frac{\Gamma \vdash e_1 : \textbf{Int}, e_2 : \textbf{Int}}{\Gamma \vdash e_1 < e_2 : \textbf{Bool}}\ (\text{INT-}<)$$

$$\frac{\Gamma \vdash T_1 \mapsto S_1 \qquad \Gamma, (X : S_1) \vdash \{\overline{s}\ e_2\} : S_2}{\Gamma \vdash \{(\textbf{type}\ X = T_1)\ \overline{s}\ e_2\} : S_2}\ (\text{BLOCK}_2)$$

Figure 2. The SimpleV2 Calculus: Type simplification and Type assignment

type decreases by one, and the left-hand side type remains unchanged. (<:-INTF-STEP) is applied until the right-hand side type is { }, and then, (<:-INTF-BASE) is applied. Note that since the **val**s are immutable, it is safe to make them covariant.

2.1.5 Why no union types?

One might ask why there are no union types in SimpleV2, because they would allow more precise typing of if-statements returning a value and also of intersection of two function types: $(T_{11} \to T_{12}) \wedge (T_{21} \to T_{22})$ could be simplified to $(T_{11} \vee T_{21}) \to (T_{12} \wedge T_{22})$. This would be more powerful than the current **union** function, which only succeeds if T_{11} is a subtype or a supertype of T_{21}.

With union types and a bottom type, the subtype relation would form a lattice where the greatest lower bound would be $\wedge$ and the least upper bound would be $\vee$, similar to the DOT declaration lattice.

The reason why no union types were added are the problems described in section 3.1. It has to be noted that at the time SimpleV2 was created, the importance of these problems was maybe a little bit overestimated.

2.1.6 Reduction rules

The next step would be to define and implement reduction rules. This was started, but they became very bulky compared to how interesting they were, and finally, the author became more interested in studying DOT than in finishing the reduction rules.

The reason why they became so bulky is that SimpleV2 is not really a core calculus containing only the features interesting for typing, but it contains many "boring" features such as binary expressions, primitive types, etc. These features were introduced to be able to write simple programs which indeed can calculate something useful, which is not really possible in DOT.

2.1.7 Conclusion

Designing and implementing the syntax and rules of SimpleV2 was very interesting and greatly helped to acquire good knowledge of PLT Redex.

Besides this, the author got an understanding of the difference between a core calculus, which can be easily implemented in PLT redex, and full-fledged languages, for which PLT redex is not the best tool to implement it. SimpleV2 was somewhere between a core calculus and a full-fledged language, so it was not clear where to continue with it, and thus, the author's attention was turned to DOT.

2.2 μ-Types

2.2.1 Motivation

A serious limitation of SimpleV2 presented in section 2.1 was that identifiers could only refer to earlier value and type declarations, and so, no recursive or mutually recursive types were possible.

However, serious practical programming needs recursive types for linked lists, for instance, and mutually recursive types for cases like, for instance, a GUI window which has a reference to its controller, which has a reference back to the window.

Secion 21.8 of [4] describes μ-Types , which are a way to define recursive types. However, only product types and function types are considered in [4].

In this section, we will show how this can be adapted to a type system with record types and function types.

2.2.2 Unfolding

μ-Types are written as $\mu X.T$, where T is a type expression which can (and typically will) contain the type variable X quantified by μ. Informally, the type $\mu X.T$ can be read as "the type X for which $X = [X \mapsto \mu X.T]T$". If we perform this substitution, we get a new type expression, which still contains $\mu X.T$, and we can again replace it by $[X \mapsto \mu X.T]T$. This process is called *unfolding* of a μ-Type.

2.2.3 Equi-recursive and iso-recurive approaches

An important difference between approaches to recursive types is the relationship between a recursive type and its one-step unfolding. In the system we are going to consider, they are equal by definition. Such approaches are called equi-recursive. In the alternative, the iso-recursive approach, a recursive type and its one-step unfolding are not considered equal, and explicit unfold and fold annotations are needed.

2.2.4 The μ-Types Calculus

The μ-Types calculus presented in this section is based on the one presented in [4], but it has record types instead of product types.

Its syntax is presented in Figure 3. The types **Pos** and **Int** were added to have some simple primitive types for which some subtype relations hold: **Pos** $<:$ **Int** $<: \top$. Note that this calculus only consists of types, and has no expressions.

2.2.5 Subtyping

The subtyping rules are algorithmic rules, and were implemented and tested with PLT redex. The four rules in the right column all are based on the same trick: In order to prove $T_1 <: T_2$, we "decompose" T_1 and/or T_2 and prove properties of these "decompositions", assuming that $T_1 <: T_2$. The intuition why this subtyping algorithm is correct comes from considering the infinite type trees associated to the μ-Types and trying to define when they are equal. The redex implementation also contains a function which prints such infinite trees up to a specified depth.

The function **cond** in the (<:-REC) rule takes two record types $\{\overline{D_i}\}$ and $\{\overline{D_j}\}$, and returns all subtype relations that must hold in order to have $\{\overline{D_i}\} <: \{\overline{D_j}\}$, or false if there's a member in $\{\overline{D_j}\}$ that is not present in $\{\overline{D_i}\}$.

Syntax

$S, T ::=$	type expression	$\Sigma ::=$	set of subtyping assumptions
X	type variable	$\{\overline{S <: T}\}$	
$\top$	Top type	l	value label in record
$S \to T$	function type	$D ::=$	declaration in record
$\{\overline{D}\}$	record type	$(\textbf{val}\ l : T)$	
$\mu X.T$	recursive type		
Pos	positive number		
Int	integer		

Subtyping $\boxed{\Sigma \vdash S <: T}$

$$\frac{(S <: T) \in \Sigma}{\Sigma \vdash S <: T}\ (\text{ASSUME}) \qquad \frac{\Sigma, (\{\overline{D_i}\} <: \{\overline{D_j}\}) \vdash \textbf{cond}(\{\overline{D_i}\}, \{\overline{D_j}\})}{\Sigma \vdash \{\overline{D_i}\} <: \{\overline{D_j}\}}\ (<:\text{-REC})$$

$$\frac{}{\Sigma \vdash T <: \top}\ (<:\text{-}\top) \qquad \frac{\Sigma, (S_1 \to S_2 <: T_1 \to T_2) \vdash T_1 <: S_1\ ,\ S_2 <: T_2}{\Sigma \vdash S_1 \to S_2 <: T_1 \to T_2}\ (<:\text{-}\to)$$

$$\frac{}{\Sigma \vdash \textbf{Pos} <: \textbf{Int}}\ (\text{POSINT}) \qquad \frac{\Sigma, (\mu X.S <: T) \vdash [X \mapsto \mu X.S]S <: T}{\Sigma \vdash \mu X.S <: T}\ (\mu\text{-}<:)$$

$$\frac{}{\Sigma \vdash T <: T}\ (\text{REFL}) \qquad \frac{\Sigma, (S <: \mu X.T) \vdash S <: [X \mapsto \mu X.T]T}{\Sigma \vdash S <: \mu X.T}\ (<:\text{-}\mu)$$

Figure 3. The μ-Types Calculus

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Sigma_3 \vdash \textbf{Pos} <: \textbf{Int}}\ \text{POSINT} \quad \dfrac{(\mu A.\{\dots\} <: \mu B.\{\dots\}) \in \Sigma_3}{\Sigma_3 \vdash \mu A.\{\dots\} <: \mu B.\{\dots\}}\ \text{ASSUME}}{\Sigma_2 \vdash \{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : \mu A.\{\dots\})\} <: \{(\textbf{val}\ head : \textbf{Int})(\textbf{val}\ tail : \mu B.\{\dots\})\}}\ <:\text{-REC}}{\Sigma_1 \vdash \{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : \mu A.\{\dots\})\} <: \mu B.\{\dots\}}\ <:\text{-}\mu}{\Sigma_0 \vdash \mu A.\{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : A)\} <: \mu B.\{(\textbf{val}\ head : \textbf{Int})(\textbf{val}\ tail : B)\}}\ \mu\text{-}<:$$

where

$\Sigma_3 = \Sigma_2\ ,\ \{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : \mu A.\{\dots\})\} <: \{(\textbf{val}\ head : \textbf{Int})(\textbf{val}\ tail : \mu B.\{\dots\})\}$

$\Sigma_2 = \Sigma_1\ ,\ (\{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : \mu A.\{\dots\})\} <: \mu B.\{\dots\})$

$\Sigma_1 = (\mu A.\{\dots\} <: \mu B.\{\dots\})$

$\Sigma_0 = \emptyset$

Figure 4. Example Derivation in μ-Types Calculus

2.2.6 Examples

An infinite stream of positive numbers can be defined as follows:

$$\mu A.\{(\textbf{val}\ head : \textbf{Pos})(\textbf{val}\ tail : A)\}$$

Since **Pos** $<:$ **Int**, the above type is a subtype of infinite integer streams, and this can be derived as shown in Figure 4.

And here's an example for mutually recursive types, where $Window$ has an id and a reference to a $Menu$, and $Menu$ has an id and a reference to a $Window$:

$$\begin{aligned}
&Window = \mu W.\{\\
&\quad (\textbf{val}\ id : \textbf{Int})\\
&\quad (\textbf{val}\ menu : \{(\textbf{val}\ id : \textbf{Int})(\textbf{val}\ window : W)\})\\
&\}\\
&Menu = \{\\
&\quad (\textbf{val}\ id : \textbf{Int})\\
&\quad (\textbf{val}\ window : Window)\\
&\}
\end{aligned}$$

2.3 EDOT: DOT as existential types

2.3.1 Intuition

Let us compare a "definition" of existential types with a "definition"[1] of DOT types.

The *existential type* $\exists\{L : S..U\}\{\overline{D}\}$ is the type inhabited by all objects *o* for which there exists a type T such that $S <: T$, $T <: U$, and *o* is of type $[T/L]\{\overline{D}\}$.

The *DOT type* $\top\{z \Rightarrow L : S..U\ ,\ \overline{D}\}$ is the type inhabited by all objects *o* for which there exists a type T such that $S <: T$, $T <: U$, and *o* is of type $\top\left\{z \Rightarrow [T/z.L]\overline{D}\right\}$.

We see that they are very similar, so it is natural to ask whether we can translate one into the other.

Note that here we are assuming that concrete types have collapsed bounds, i.e. that all the type members of all objects also have collapsed bounds (see also section 3.2 for more details).

2.3.2 Syntax

To get started, we establish a small "EDOT Calculus" [2], which is similar to the DOT Calculus presented in [2] but does not have union and intersection types, and only allows refinements of $\top$, which it represents as existential types. Its syntax is presented in Figure 5. It doesn't have reduction rules, and the type assignment rules are the same as in the DOT Calculus.

No distinction between abstract and concrete types is needed, because the only way to create new objects is to instantiate a refinement of $\top$. This could be considered as a typed version of Javascript's way of constructing objects.

Existential types can quantify over several type variables at once. This allows the bounds of a type variable X_1 to refer to a type variable X_2, while the bounds of X_2 can also refer to X_1.

In the existential type $\exists\{\ \overline{(X\ L\ S\ U)}\ \}\{\overline{D}\}$, X is the name of the type variable, so inside $\overline{D}$, X is bound. S and U are the lower and upper bound of X. The label L is used to mark the "role" of the type variable: When two existential types are compared to see if the first is a subtype of the second, the subtyper can know which type variable in the first type corresponds to which type variable in the second type by comparing the labels of the type variables, because two type variables only correspond to each other if they have the same label.

Note that we cannot simply use the type variable X for this, because in nested existential types with type variables having the same role, we want the body of the inner existential type to be able to refer to the type variables of the outer existential type, which would be shadowed if we had only type variables and no type labels.

2.3.3 Translation from DOT to EDOT

We can translate a subset of DOT types and expressions into EDOT.

To translate types, $\bot$ and $\top$ are left unchanged, and refinements of $\top$ are translated into an existential type, with each type member becoming a type variable quantified by $\exists$. Path types are translated into type variables. This is only possible if the path of the path type is of length one, because otherwise the type variable corresponding to the path type is not bound in the scope we're in. This is a significant restriction on the expressive power of EDOT.

To translate expressions, we only need to translate all types they contain.

2.3.4 Translation from EDOT to DOT

Translation from EDOT to DOT should be possible, because we believe that EDOT is a strict subset of DOT.

2.3.5 Subtyping

The subtyping rules of EDOT are presented in Figure 5. They are algorithmic rules, and the subtyper has to apply them in the order R_1, R_2, ... R_7, always choosing the first rule that matches. Note that R_4 gives a negative result ($\not\vdash$).

The operator $\oplus$ updates B with new bounds, i.e. if B already contains bounds for the type variable to be added, they are overridden, otherwise the new bounds are just added. Note that here, type variables are compared, and not the type labels.

The operator $\triangleleft$ "synchronizes" the names of type variables: $E_b \triangleleft E_a$ returns E_b, but all type variables of E_b which have a label that also appears in the type variables of E_a are renamed such that that they have the same name as in E_a.

[1] "Definition" is quoted because it's in an informal style, which is sufficient here, because the purpose is to describe the intuition, which will be made precise later.

[2] The "E" stands for "existential types"

Syntax

x, y, z	Variable	$L ::= L_t$	Type label
l	Value label	$S, T, U, V, W ::=$	Type
m	Method label	$\quad X$	type variable
$v ::=$	Value	$\quad E$	existential type
$\quad x$	variable	$\quad \top$	top type
$t ::=$	Term	$\quad \bot$	bottom type
$\quad v$	value	$D ::=$	Declaration
$\quad$ **val** $x =$ **new** $c;\ t$	new instance	$\quad l : T$	value declaration
$\quad t.l$	field selection	$\quad m : S \to U$	method declaration
$\quad t.m(t)$	method invocation	$E ::=$	Existential type
$c ::=$	Constructor	$\quad \exists\{\ \overline{(X\ L\ S\ U)}\ \}\{\overline{D}\}$	
$\quad \top\ \{z \Rightarrow \overline{D}\}\ \{\overline{d}\}$	$\overline{D}$ and $\overline{d}$ must match	$b ::=$	Bounds of a type variable
$d ::=$	Initialization	$\quad X : S..U$	
$\quad l = v$	field initialization	$B ::=$	Set of bounds
$\quad m(x) = t$	method initialization	$\quad \overline{b}$	

Subtyping

$$\frac{}{B \vdash T <: \top} \quad (\mathrm{R}_1)$$

$$\frac{}{B \vdash \bot <: T} \quad (\mathrm{R}_2)$$

$$\frac{}{B \vdash T <: T} \quad (\mathrm{R}_3)$$

$$\frac{\mathbf{cond}(E_b, E_a) = \mathbf{false}}{B \not\vdash E_b <: E_a} \quad (\mathrm{R}_4)$$

$$\frac{B \oplus \{\overline{(X_b\ L_b\ S_b\ U_b)} \triangleleft E_a\} \vdash \mathbf{cond}(\exists\{\overline{(X_b\ L_b\ S_b\ U_b)}\}\{D_b\} \triangleleft E_a, E_a)}{B \vdash \exists\{\overline{(X_b\ L_b\ S_b\ U_b)}\}\{D_b\} <: E_a} \quad (\mathrm{R}_5)$$

$$\frac{B \vdash U <: T \qquad (S <: U) \in \mathbf{trcl}(\mathbf{rel}(B))\ ,\ S \neq U}{B \vdash S <: T} \quad (\mathrm{R}_6)$$

$$\frac{B \vdash S <: U \qquad (U <: T) \in \mathbf{trcl}(\mathbf{rel}(B))\ ,\ U \neq T}{B \vdash S <: T} \quad (\mathrm{R}_7)$$

Figure 5. The EDOT Calculus

$$\dfrac{\dfrac{}{B_1 \vdash \bot <: X_C}\,\mathrm{R}_2 \quad \dfrac{(X_C <: X_D) \in \mathbf{trcl}(\mathbf{rel}(B_1))}{B_1 \vdash X_C <: X_D}\,\mathrm{R}_6 \quad \dfrac{}{B_1 \vdash \bot <: X_D}\,\mathrm{R}_2 \quad \dfrac{(X_D <: X_C) \in \mathbf{trcl}(\mathbf{rel}(B_1))}{B_1 \vdash X_D <: X_C}\,\mathrm{R}_6 \quad \dfrac{(X_C <: X_D) \in \mathbf{trcl}(\mathbf{rel}(B_1))}{B_1 \vdash X_C <: X_D}\,\mathrm{R}_6}{B_0 \vdash \exists\{(X_C\ C \bot X_D)(X_D\ D \bot X_C)\}\{x : X_C\} <: \exists\{(X_C\ C \bot X_D)(X_D\ D \bot X_C)\}\{x : X_D\}}\,\mathrm{R}_5$$

where

$$B_1 = B_0 \cup \{X_C : \bot..X_D\ ,\ X_D : \bot..X_C\}$$
$$B_0 = \emptyset$$

Figure 6. Example Derivation in EDOT Calculus (using simplified form of R_6)

When a type variable is renamed, a substitution in the body of the existential type is performed, of course. For simplified notation, the $\triangleleft$ operation can be applied not only on an existential type, but also just on a set of bounds.

Rule R_5 compares two existential types for subtyping. Informally speaking, it does the following: First, it "synchronizes" the names of the type variables using the $\triangleleft$ operator, then it assumes the bounds of the left-hand side type and tries to prove the bounds of the right-hand side, as well as the subtype relationships needed for the value and method members.

To make this more formal, we define the **cond** function, which takes two existential types E_b and E_a and returns a set of tuples $\overline{S <: T}$, the "subtype conditions" that need to be satisfied in order to have $E_b <: E_a$, or if there is a type variable, value label or method label in E_a that is not in E_b, it returns **false**.

For each type variable X in E_a, with bounds $S..U$, the subtype conditions include $S <: X$ and $X <: U$, for each value label $l : T_a$ in E_a, with a corresponding value label $l : T_b$ in E_b, the subtype conditions include $T_b <: T_a$, and for each method label $m : V_a \to W_a$ in E_a, with a corresponding method label $m : V_b \to W_b$ in E_b, the subtype conditions include $V_a <: V_b$ (inverted because of contravariance) and $W_b <: W_a$.

Notice that the way type labels are treated w.r.t. subtype conditions is different from what readers of [2] might expect: It is *not* the case that for each type variable X in E_a with bounds $S_a..U_a$, and with a corresponding type variable X in E_b with bounds $S_b..U_b$, the subtype conditions include $S_a <: S_b$ and $U_b <: U_a$, because that differs from what's defined above. This difference will be important later, and we will refer to it by **Trick 1**.

If a subtype relationship between holds in DOT, it also holds in EDOT, because in DOT, this means that the bounds of the left-hand side types are narrower, and if we suppose these bounds, we can always prove the looser bounds in the right-hand side type. Whether the converse implication holds is more difficult to say and prove, because it depends very much on how DOT expansion is implemented.

The function **rel** takes a set of bounds, and turns it into a set of subtype tuples:

$$\textbf{rel}(\overline{X : S..U}) = \{\overline{S <: X}\ ,\ \overline{X <: U}\}$$

The function **trcl** calculates the transitive closure of a relation.

Contrary to DOT, when one argument in the $S <: T$ judgement to prove or disprove is a type variable (corresponds to a path type in DOT), EDOT does not just replace the path type by its upper bound, which can lead to infinite loops if the graph of the subtype relation among path types contains cycles. Instead, it calculates the transitive closure of the subtype relation over type variables (using matrix multiplications, which never runs into infinite loops) and checks if a desired subtype judgement is in it. This is done in R_6 and R_7, which also include a "hardcoded" transitivity rule, because the transitive closure only contains type variables, but T (in R_6) or S (in R_7) might not be a type variable. We will refer to this by **Trick 2**.

2.3.6 Examples

In the following example, the $z.L$ in T_1 does not have an expansion in DOT (in the current version of DOT, it does, but we're discussing [2] here). In the right column, the corresponding EDOT type is presented:

$$\begin{array}{l|l} T_1 = \top\{z \Rightarrow & T_1 = \exists\{(X_L\ L \perp X_L)\}\{ \\ \quad L : \perp..z.L & \quad l : X_L \\ \quad l : z.L & \} \\ \} & \end{array}$$

We are going to compare the above T_1 with the following type T_2, which has no expansion problems in DOT:

$$\begin{array}{l|l} T_2 = \top\{z \Rightarrow & T_2 = \exists\{(X_L\ L \perp \top)\}\{ \\ \quad L : \perp..\top & \quad l : X_L \\ \quad l : z.L & \} \\ \} & \end{array}$$

Clearly, we should have $T_1 <: T_2$, because $z.L <: \top$. However, DOT fails to prove this, because to have $z.L <: \top$, $z.L$ must be **wfe**, but $z.L$ has no expansion.

In EDOT, $T_1 <: T_2$ can be derived as follows:

$$\dfrac{\dfrac{}{\vdash \perp <: X_L}\,R_2 \quad \dfrac{}{\vdash X_L <: \top}\,R_1 \quad \dfrac{}{\vdash X_L <: X_L}\,R_3}{\vdash \exists\{(X_L\ L \perp X_L)\}\{l : X_L\} <: \exists\{(X_L\ L \perp \top)\}\{l : X_L\}}\,R_5$$

Further, in EDOT, we also have $T_2 <: T_1$, because $z.L$ contains no declarations at all, so $\top <: z.L$. In DOT, however, we don't have $T_2 <: T_1$, because it would not work with refinements.

Another example for a derivation is presented in Figure 6.

2.3.7 Conclusion

Trick 1, i.e. to use the subtype relations guaranteed in the left-hand side type of $<:$ as assumptions to prove the subtype relations needed in the right-hand side type, combined with **Trick 2**, i.e. to use the concept of the transitive closure of a relation, results in a type system which *does not need expansion* and can deal with cyclic subtype relation graphs in a simple and elegant way.

2.4 Trying to generalize EDOT to DOT

Now it is natural to ask whether the ideas from EDOT can be generalized to DOT.

The first approach was to design a function F which takes a Γ, and using the type of each variable, recursively collects all paths types that exist according to Γ, and then returns for each path type $p.L$ two tuples $S <: p.L$ and $p.L <: U$, where S is the lower and U the upper bound of $p.L$.

$F(\Gamma)$ could then be used in a similar way as the B in EDOT.

However, even in cases with no cyclic subtype graphs, this approach does not work, as illustrated with the generic recursive lists example below:

$$
\begin{array}{l}
\Gamma = \{(w : \top\{w \Rightarrow \\
\quad L : \bot..\top\{l \Rightarrow \\
\qquad T : \bot..\top \\
\qquad head : l.T \\
\qquad tail : w.L\,\{w \Rightarrow T : \bot..l.T\} \\
\quad \} \\
\}), \\
(l : w.L)\}
\end{array}
$$

The path types that F would have to consider are $l.T$, $l.tail.T$, $l.tail.tail.T$, and so on. We see that there are infinitely many of them, so F will never terminate.

Since this seems to be quite a deep problem, no further attempts to generalize EDOT to DOT were made, and instead, it was investigated whether DOT can handle all cases where such "infinite paths" occur (see next section).

2.5 Infinite paths

There are examples where the DOT typechecking algorithm encounters path types of ever growing length and thus never terminates. We will refer to this phenomenon by "infinite paths", even though each path is of finite length, of course.

We are using the current[3] PLT Redex implementation of DOT, and the current[4] rules description.

All examples have been implemented and checked in PLT Redex.

In this section, we will discuss such infinite paths. Note that this has nothing to do with the example presented in section 4.2 of [2]. There, the problem is that the graph of the subtype relation contains cycles, whereas the problem with infinite paths is that the graph of the subtype relation is not finite.

[3] `https://github.com/namin/dot/blob/5b3b490/src/redex/dot.rkt` from May 22, 2013

[4] `https://github.com/namin/dot/blob/2abbb0c/doc/current/rules.tex` from June 3, 2013

2.5.1 Infinite loop in typechecking

Let

$$
\begin{array}{l}
T_1 = \top\{a \Rightarrow \\
\qquad L : \bot..\top\{z \Rightarrow \\
\qquad\quad M : z.f.M..z.f.M \\
\qquad\quad f : a.L \\
\qquad \} \\
\qquad m : \top \to a.L \\
\quad \}
\end{array}
$$

Now if we want to check if the expression

$$
\begin{array}{l}
\textbf{val}\ a = \textbf{new}\ T_1\{ \\
\quad m(x) = \textbf{val}\ r = \textbf{new}\ a.L\{f = r\};\ r \\
\};\ a
\end{array}
$$

is of type T_1 (using the $:_{<:}$ judgement), the algorithm runs into an infinite loop. Let us analyze this:

It has to typecheck the **new** $a.L$, so it checks that all type members of $a.L$ are **wf** (in this case, that's only M, which becomes $r.M$ in the context).

In previous versions of DOT, even this failed, because the algorithm checked that the upper bound of $r.M$, which is $r.f.M$, is **wf**, which required to check that $r.f.f.M$ is **wf**, and so on.

In the current version of DOT, the **wf** rules have been relaxed, so that the bounds are not recursively checked for well-formedness any more, but this does not solve the problem: After checking that the bounds are **wf**, the algorithm continues by checking that the lower bound is a subtype of the upper bound, i.e., that $r.M <: r.M$. To do so, it tries to apply the REFL rule, but this requires that $r.M$ is **wfe**, so it has to expand $r.M$, which requires to expand the upper bound of $r.M$, which is $r.f.M$, so $r.f.f.M$ has to be expanded, and so on.

The algorithm could be changed so that instead of applying the REFL rule, it would apply either the <:-TSEL rule or the TSEL-<: rule, but this would also lead to an infinite loop.

Now the question arises whether this is a problem of subtyping or a problem of expansion. Let $\Gamma = \{(a : T_1), (r : a.L)\}$ and $s = \emptyset$. Then, when we apply the expansion algorithm to find the $\overline{D}$ for which $\Gamma; s \vdash r.M \prec_z \overline{D}$, it does what's described above, i.e. does not terminate, so it is a problem of expansion.

2.5.2 A workaround: The "magic" function

An alternative expansion algorithm was created which solves the problem described above. It is based on the expansion algorithm from [2], i.e. it does not calculate a fixed point. The trick is that it uses a so-called "magic" function:

Definition: A function $f : \overline{\Gamma} \times \overline{s} \times \overline{T} \longrightarrow \mathbb{N}$ is called magic if for all Γ, s, T, $\overline{D}$, we have $\Gamma; s \vdash T \prec_z^{n.a.} \overline{D}$

implies that in the derivation of $\Gamma; s \vdash T \prec_z^{n.a.} \overline{D}$, no path types of length greater than $f(\Gamma, s, T)$ appear.

The $\prec^{n.a.}$ sign means expansion according to the rules of [2], without any loop avoidance tricks, so these are non-algorithmic rules.

In order to expand type T, the algorithm first calls the magic function to get $n := f(\Gamma, s, T)$, and during expansion, whenever a type S would expand to a path type of length greater than n, S is expanded to the empty set of declarations. It is safe to do so because f guarantees that we needn't look at path types of length greater than n.

It is not yet clear whether there is such a magic function which is computable, nor how it could be implemented.

To write a preservation proof, one could simply assume that f always returns correct results, and in practice, one would set the return value of f to a large arbitrary constant. However, whatever constant we choose, there is no guarantee that it is large enough.

3. Side topics

3.1 Types and mathematical sets

To each type T, we can associate two sets: The set $\overline{D}$ of *declarations* that it contains (this is expressed as $\Gamma; s \vdash T \prec_z \overline{D}$ in DOT) and the set Ω of *objects* which are of type T (this could be written as $\Omega(T) := \{c \mid \Gamma; s \vdash o :_{<:} T\}$, combining set notation and DOT notation). In this section, we will consider the sets of objects associated to types.

Most programmers are familiar with mathematical sets, set union and set intersection. So, it would be desirable to have a programming language where we can say "types just behave as mathematical sets". Formally, this can be expressed as

$$T_1 <: T_2 \quad \Leftrightarrow \quad \Omega(T_1) \subseteq \Omega(T_2) \tag{1}$$

The $\Rightarrow$ direction of (1) holds in all "normal" type systems, because without it, preservation cannot be proved without substantially changing the semantics of subtyping. Actually, it is a direct consequence of the subsumption rule:

$$\frac{\Gamma \vdash x : T_1 \ , \ \Gamma \vdash T_1 <: T_2}{\Gamma \vdash x : T_2} \ (\textsc{Subsumption})$$

The $\Leftarrow$ direction of (1) is not strictly necessary for a type system to be sound, but type system designers should try to make it hold for as many cases as possible, because each case where it doesn't hold corresponds to a program snippet that should typecheck according to the programmer's intuition, but is rejected by the typechecker, and this is perceived by the programmer as a flaw of the typesystem.

If a language provides union and intersection types, it becomes more difficult to guarantee that (1) holds. Let us consider all subtyping statements that must hold in order to have (1):

$$T_1 \vee T_2 <: T \quad \Leftrightarrow \quad \Omega(T_1) \cup \Omega(T_2) \subseteq \Omega(T) \tag{2}$$

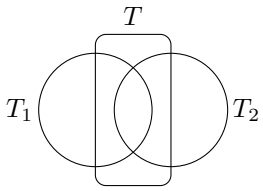
$$T_1 \wedge T_2 <: T \quad \Leftrightarrow \quad \Omega(T_1) \cap \Omega(T_2) \subseteq \Omega(T) \tag{3}$$

$$T <: T_1 \wedge T_2 \quad \Leftrightarrow \quad \Omega(T) \subseteq \Omega(T_1) \cap \Omega(T_2) \tag{4}$$

$$T <: T_1 \vee T_2 \quad \Leftrightarrow \quad \Omega(T) \subseteq \Omega(T_1) \cup \Omega(T_2) \tag{5}$$

(2) and (4) are easy to implement, because it suffices to check that both T_1 and T_2 are subtypes of T for (2), and that T is a subtype of both T_1 and T_2 for (4). This corresponds to the $\vee$-$<:$ and $<:$-$\wedge$ rules in DOT.

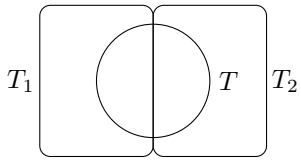
(3) is more difficult to implement, because of situations like the one depicted by the following Venn diagram:



Here, it is not sufficient to just check whether T_1 is a subtype of T or T_2 is a subtype of T, but that's what the $\wedge_1$-$<:$ and $\wedge_2$-$<:$ rules of DOT do, so one might expect to find an example in DOT where such a situation causes the $\Leftarrow$ part of (1) to be violated.

However, there was not (yet?) found any such example, because in all situations which might seem problematic at first sight, the $<:$-RFN rule is applied eventually, so $T_1 \wedge T_2$ is expanded to a set of declarations, and subtyping behaves as expected.

(5) also is more difficult to implement, because of situations like this:



Here, it is not sufficient to just check whether T is a subtype of T_1 or T is a subtype of T_2. But that's all that the DOT subtyper does (rules $<:$-$\vee_1$ and $<:$-$\vee_2$), and that's why there's an example where (1) does not hold in DOT: Let A and B stand for any two disjoint types, and consider

$T = \{z \Rightarrow l : A \vee B\}$
$T_1 = \{z \Rightarrow l : A\}$
$T_2 = \{z \Rightarrow l : B\}$

Here, T is not a subtype of $T_1 \vee T_2$ according to the current DOT subtyper, even though it would be safe to say so.

However, as [1] pointed out, this cannot be generalized to a similar example with several such members without getting a very long right-hand side, as illustrated by the case where we have two such members:

$T = \{z \Rightarrow\ ,\ l_1 : A \vee B\ ,\ l_2 : A \vee B\}$
$T_1 = \{z \Rightarrow l_1 : A\ ,\ l_2 : A\}$
$T_2 = \{z \Rightarrow l_1 : A\ ,\ l_2 : B\}$
$T_3 = \{z \Rightarrow l_1 : B\ ,\ l_2 : A\}$
$T_4 = \{z \Rightarrow l_1 : B\ ,\ l_2 : B\}$

Here, the subtyping statement which cannot be proved by the current DOT subtyper is $T <: T_1 \vee T_2 \vee T_3 \vee T_4$.

3.2 Non-collapsed bounds in concrete types

Let us call the bounds of a DOT type label *collapsed* if the lower bound is the same as the upper bound, and *non-collapsed* if the lower bound is a strict subtype of the upper bound.

In abstract types, allowing non-collapsed bounds is crucial, because otherwise, type members would not me more powerful than simple type aliases, which save some rewriting, but do not really add much power to the type system.

In concrete types (i.e. types which can be instantiated with **new**), however, one might ask if non-collapsed bounds are really necessary.

Let's consider this example (A and B stand for any types, and $A <: B$):

$$\begin{aligned}&\textbf{val}\ z = \textbf{new}\ \top\{z \Rightarrow L : A..B\ ,\ m : z.L \to z.L\}\{\\ &\quad m(x) = \langle\text{body}\rangle\\ &\};\ z\end{aligned}$$

It is "complete" in the sense that it contains a type label in a covariant position, as well as a type label in a contravariant position, so the observations made here can be generalized to any other example.

In order for this example to typecheck, $\langle$body$\rangle$ must be such that it accepts any $t :_{<:} B$, and it must return a $t :_{<:} A$.

But this means that the example would also typecheck if we had $L : B..B$ instead of $L : A..B$. The only difference is that users of z now can feed any $t :_{<:} B$ to $z.m$, whereas before, they had to provide a $t :_{<:} A$. If this difference is not desired, it suffices to up-cast z to $\top\{z \Rightarrow L : A..B\ ,\ m : z.L \to z.L\}$, to get exactly the same behaviour as in the original example, but without using non-collapsed bounds in the type of the constructor.

So, we can conclude that if the type in a constructor is a refinement of $\top$, the restriction that its type members must have collapsed bounds does not remove any power from the language.

In the context of DOT, this conclusion might not be very interesting, but in the context of purely structurally typed languages, it is, because there, all objects are created in an "ad hoc" way, i.e. without specifying a type in the constructor, but by simply enumerating all members, together with their type and their initialization. This can be rewritten as a DOT type constructor whose type type is a refinement of $\top$, putting the types of the members in the refinement, and the initializations in the initializations block of the DOT constructor.

This means that in order to introduce path dependent types in the SimpleV2 language (see section 2.1), only the definition of interface types would have to be changed, and the definition of object constructions could remain unchanged, i.e. type members would have the form (**type** $X = T$), and not (**type** $X : S..U$), without loosing any power.

3.3 Bugs detected

3.3.1 Declaration intersection in Redex model

There was a bug in the Redex model of DOT, which affected declaration intersections: $\{l_1 : \top\} \wedge \{l_1 : \top\}$ was evaluated to $\{l_1 : \top \wedge \top\}$ instead of just $\{l_1 : \top\}$.

Additionally, $\{l_A : \top\ ,\ l_C : \top\} \wedge \{l_B : \top\ ,\ l_C : \bot\}$ was evaluated to $\{l_A : \top\ ,\ l_C : \top\ ,\ l_B : \top\ ,\ l_C : \bot\}$ instead of $\{l_A : \top\ ,\ l_B : \top\ ,\ l_C : \bot\}$.

More confused than convinced of having found a bug, the author reported this behaviour to N. Amin, who fixed the problems in commit `95299b2`.

3.3.2 Wfe type not a subtype of $\top$ in Scala model

In the Scala model of DOT, the following example yielded the error "$a.S$ is not a subtype of $\top$", instead of passing typechecking:

$$\begin{aligned}&\textbf{val}\ w = \textbf{new}\ \top\{y \Rightarrow\\ &\quad f : \top\{a \Rightarrow S : \bot..\top\ ,\ l : a.S\} \to \top\\ &\}(f(a) = a.l);\ w\end{aligned}$$

The author tried to find and fix this bug, but didn't succeed, so he just added a testcase illustrating the problem and reported it to N. Amin, who fixed it in commit `7d0d3c2`. The problem was that when the body of a method with argument x of type T was typechecked, the environment was not updated with the binding $(x : T)$.

4. Conclusion

In all classes and projects I have taken so far at EPFL, it was precisely specified what we had to study, and what deliverables we had to create for the projects. When I was interested in topics which were not on the predefined curriculum, there was only little time left besides to study them, and no mentoring or guidance was provided for the study of such topics.

The highly appreciated exception to that was this semester project. I was free to investigate any topic in type systems I was interested in (or at least, that's the impression I had and enjoyed).

Regular meetings with the supervisor, Nada Amin, included valuable discussions and hints on where to continue.

Contrary to all other projects done at EPFL so far, I was not required to hand in a well-specified "product" at the end of the semester. Instead, the "product" is a collection of various insights on type systems, and the most important of them are described in this report.

One might criticize the lack of a well-specified "product", but personally, I found the freedom that I was granted in this project much more valuable than having a "product" at the end.

References

[1] N. Amin. Semester project meetings. 2013.

[2] N. Amin, A. Moors, and M. Odersky. Dependent object types. 2012.

[3] C. Klein, J. Clements, C. Dimoulas, C. Eastlund, M. Felleisen, M. Flatt, J. A. McCarthy, J. Rafkind, S. Tobin-Hochstadt, and R. B. Findler. Run your research: on the effectiveness of lightweight mechanization. In *POPL*, pages 285–296, 2012.

[4] B. C. Pierce. *Types and programming languages*. MIT Press, 2002. ISBN 978-0-262-16209-8.