Dependent Object Types With Existential Quantification Over Objects

Samuel Grütter
EPFL
samuel.gruetter@epfl.ch

Abstract

We present exDOT, a new variant of the Dependent Object Types (DOT) calculus featuring existential quantification over objects.

Further, we define a subset of the exDOT calculus called gDOT, which does not have existentials, but whose type safety proof still poses some interesting challenges.

We prove type safety for gDOT and discuss the difficulties in extending the proof to the full exDOT calculus.

1. The exDOT and gDOT calculi

This section presents the exDOT and gDOT calculi. The latter is a strict subset of the former, obtained by removing existential types and singleton types, but it still features recursive types and recursive methods.

Figures 1 to 6 present the syntax and the rules. The parts highlighted in gray are those which only belong to exDOT, but not to gDOT.

1.1 Notation

We use $\overline{X}$ to denote a set of X's, and X_i to denote an arbitrary element of that set. Moreover, we sometimes write $\Gamma \vdash P\ ,\ Q$ instead of "$\Gamma \vdash P$ and $\Gamma \vdash Q$".

1.2 Syntax

Figure 1 presents the syntax. Variables x, y, z, which are freely alpha-renamable, are used to name the argument of methods, to name the object over which existential types $\exists(x : T)U$ quantify, as binders to name allocated objects in **val** x = **new** $\{\overline{d}\};\ t$, and they are also used to denote locations in the store (heap) during reduction.

A second alphabet, disjoint from the variables, is used to name type members L of objects and types, and a third disjoint alphabet is used to name method members m of objects and types. The symbol l is used to denote a type label or method label. Contrary to variables, type labels and method labels are not alpha-renamable.

A term can be a variable x, or an object creation followed by a term, **val** x = **new** $\{\overline{d}\};\ t$, where a list of type-annotated initializations is given to construct a new object. Further, a term can also be a method invocation $t.m(u)$, where t is the receiver, m the method name and u is the argument. Note that all methods take exactly one argument, because methods without arguments can be encoded by passing unit, and multi-argument methods can be encoded using tuples.

Initializations d are used to construct objects. They are either a type definition $L = T$, or a method definition $m(x : T) : U = u$.

A typechecking environment Γ maps variables to types. The order of its mappings does not matter, and the type T_i of a mapping $x_i : T_i$ may contain any variable mapped by the environment, including x_i itself. This will allow a restricted form of recursive types without any need for a special type constructor.

A store s models the heap during program execution, and it maps variables to objects $\{\overline{d}\}$. Contrary to typechecking environments, the order of the mappings in a store does matter; it corresponds to the order in which the objects were allocated. The definitions $\overline{d}$ of a mapping $x \mapsto \{\overline{d}\}$ may contain any variable defined earlier in the store, but no variables defined later. References to x itself are allowed, and this will enable recursive methods (and recursive type definitions as well).

Types can be the top type $\top$, which is inhabited by all objects, or the bottom type $\bot$, which is the uninhabited type. Terms of type $\bot$ can only be obtained by infinite recursion. $\{D\}$ denotes a record type with one declaration. In order to obtain record types of several declarations, type intersection can be used. A type selection $x.L$ denotes the type member L of object x, and the intersection type $T_1 \wedge T_2$ is inhabited by all objects which inhabit both T_1 and T_2, whereas the union type $T_1 \vee T_2$ is inhabited by all objects inhabiting T_1 or T_2.

Moreover, exDOT extends gDOT by adding the singleton type x.**type**, which is inhabited only by x, and the existential type $\exists(x : T)U$, which abstracts over an object x (of type T) on which U depends. Note that not only U can refer to the x bound by $\exists$, but also T. This will allow recursive types.

Declarations D are the basic building blocks to create types. The abstract type declaration $L : S..U$ declares an abstract type member which is upper-bounded by U and

Syntax

x, y, z	Variable	$S, T, U, V, W ::=$	Type
$l ::=$	Label	$\top$	top type
L	Type label	$\bot$	bottom type
m	Method label	$\{D\}$	one-member record
$t, u ::=$	Term	$x.L$	type selection
x	variable	$T \wedge T$	intersection type
val $x =$ **new** $\{\overline{d}\};\ t$	new instance	$T \vee T$	union type
$t.m(u)$	method invocation	$x.$**type**	singleton type
$d ::=$	Initialization	$\exists(x : T)U$	existential type
$L = T$	type definition	$D ::=$	Declaration
$m(x : T) : U = u$	method definition	$L : S..U$	abstract type decl.
$\Gamma ::= \overline{x : T}$	Environment	$m : S \to U$	method declaration
$s ::= \overline{x \mapsto \{\overline{d}\}}$	Store		

Reduction $\boxed{t \,|\, s \ \to\ t' \,|\, s'}$

$$\frac{x \mapsto \left\{\overline{L = W}\ \overline{m(z : T) : U = u}\right\} \in s}{x.m_i(y) \,|\, s \ \to\ [y/z_i]u_i \,|\, s} \quad \text{(RED-CALL)}$$

$$\frac{z \notin \mathit{dom}(s)}{\textbf{val}\ x = \textbf{new}\ \{\overline{d}\};\ t \,|\, s \ \to\ [z/x]t \,|\, s, z \mapsto [z/x]\{\overline{d}\}} \quad \text{(RED-NEW)}$$

$$\frac{t \,|\, s \ \to\ t' \,|\, s'}{t.m(u) \,|\, s \ \to\ t'.m(u) \,|\, s'} \quad \text{(RED-CALL-1)} \qquad \frac{u \,|\, s \ \to\ u' \,|\, s'}{x.m(u) \,|\, s \ \to\ x.m(u') \,|\, s'} \quad \text{(RED-CALL-2)}$$

Figure 1. The exDOT Calculus: Syntax and Reduction rules

lower-bounded by S. That is, a type definition $L = T$ only conforms to it if T is a subtype of U and a supertype of S. The method declaration $m : S \to U$ declares a method taking an argument of type S and returning a result of type U.

Previous versions of the DOT calculus (e.g. [1]) also included value members of the form $l : T$, but this can be modeled by a method taking unit and returning T. The reason this was not done in [1] is that their selection types are of the form $p.L$, where p can be a path of several field selections, which could not be modeled by methods, because the return value of methods is not considered stable. In this paper, however, we restrict selection types to the form $x.L$, so adding value members $l : T$ does not add any expressivity.

1.3 Rules overview

The reduction rules $t \,|\, s \ \to\ t' \,|\, s'$ presented in Figure 1 don't depend on any other rules.

In order to define the typing rules, we have to start by defining declaration $intersection$ and $union$. Then we can define the two mutually recursive membership ($\Gamma \vdash T \ni D$) and non-membership ($\Gamma \vdash T \not\ni D$) judgments, which only depend on declaration intersection and union. This is presented in Figure 3.

Next, we can define the well-formedness judgments $\Gamma; \overline{W} \vdash T$ **wf** and $\Gamma; \overline{W} \vdash D$ **wf**. These two mutually recursive judgments, presented in Figure 6, only depend on the membership judgment.

Given these definitions, we can define subtyping (Figure 5). It consists of the two mutually recursive judgments $\Gamma \vdash S <: T$ and $\Gamma \vdash D <: D'$.

Finally, we can define term typing (Figure 2), which consists of the three mutually recursive judgments $\Gamma \vdash t : T$, $\Gamma \vdash d : D$, and $\Gamma \vdash \{\overline{d}\} : T$, and depends on all previously defined judgments.

For the presentation in this paper, however, we don't follow this bottom-up order, but we chose a different order which is hopefully easier to understand.

1.4 Reduction rules

The small-step reduction rules follow call-by-value semantics. The judgment $t \,|\, s \ \to\ t' \,|\, s'$ states that performing

one evaluation step on term t in store s yields term t' in the (possibly extended) store s'.

The final result of evaluating a term is always of the form $x|s$, where x is bound in s.

RED-CALL invokes a method by substituting the argument, which is required to be a variable, in the body of the method definition. Note that if we had full beta reduction, i.e. if we could also substitute a not yet fully evaluated term, the substitution might turn a $z_i.L$ type into something of the form $t.L$, which does not even conform to the grammar.

RED-NEW adds the newly constructed object to the store, making sure its name z is fresh.

The remaining two rules, RED-CALL-1 and RED-CALL-2, are just congruence rule allowing reduction in subterms.

1.5 Term typing rules

The initialization typing judgment $\Gamma \vdash d : D$ typechecks type and method member definitions. It assigns to a declaration d exactly the type that d declares, so it's precise and unique (contrary to the term typing judgment).

The initialization list typing judgment $\Gamma \vdash \{\overline{D}\} : T$ assigns $\top$ to the empty list of definitions and an intersection type to non-empty lists of definitions. By $label(d)$, we mean L if $d = (L = T)$, and m if $d = (m(x : T) : U = u)$, and by $labels(\overline{d})$, we mean the set obtained by applying $label$ to all definitions in $\overline{d}$.

The term typing judgment $\Gamma \vdash t : T$ assigns types to terms, and does so in an imprecise (and thus non-unique) way, because of the subsumption rule TY-SBSM. Note that in TY-NEW, $\{\overline{d}\}$ is typechecked with the self reference x in the environment, which enables recursive definitions.

The most interesting rule is the skolemization rule TY-SKOLEM: If we have an x in the environment whose type depends on some unknown y, it allows us to assume such a y to typecheck the term. At the end, however, y must not appear in t nor in T, because otherwise it would be unbound in the conclusion. But note that y may appear in all intermediate steps of the $\Gamma, x : U, y : S \vdash t : T$ proof.

The term typing judgment is regular with respect to well-formedness, i.e. we can prove trivial lemmas stating that each type and each declaration occuring in the conclusion of a typing judgment is well-formed. This property will simplify the type safety proof considerably.

1.6 Declaration intersection and union

Figure 3 presents $intersect$ and $union$. These functions take two declarations whose labels must be the same, and return the intersection (or union, respectively) of the two declarations. Notice that method arguments and lower bounds are contravariant positions, so intersection and union are swapped there.

1.7 Membership and non-membership

The membership judgment $\Gamma \vdash T \ni D$ states that type T has declaration D. It is plays the same role as the expansion judgment $\Gamma \vdash T \prec_z \overline{D}$ in [1], but only focuses on one declaration at a time instead of giving all of them.

The soundness proof requires that membership be unique, i.e. that given Γ, T and a label l, there is at most one declaration D such that $\Gamma \vdash T \ni D$ and $label(D) = l$.

In order to achieve this, we have to define an auxiliary non-membership judgment $\Gamma \vdash T \not\ni l$, so that the $\wedge$-$\ni$-1 and $\wedge$-$\ni$-2 rules only can discard one side of the intersection type if it contains no declaration with the label in question.

Note that to prove non-membership in a union type $T_1 \vee T_2$, it would be sufficient to prove non-membership in T_1 or T_2, so two rules (instead of three) would be sufficient. However, for the narrowing proof, which goes by induction over non-membership derivations, it is more convenient if the derivation always explores both sides of the union type, so that we have an inductive hypothesis for both sides. That's why there are the three rules $\vee$-$\not\ni$-1, $\vee$-$\not\ni$-2, and $\vee$-$\not\ni$-12 instead of just two.

```
lookup(G, T, l) := T match {
  case ⊤ ⇒ None
  case ⊥ ⇒ l match {
    case L ⇒ (L : ⊤..⊥)
    case m ⇒ (m : ⊤ → ⊥)
  }
  case {D} ⇒ if label(D) = l then D else None
  case x.L ⇒
    let T := Γ(x) in
    let (L : S..U) := lookup(G, T, L) in
    lookup(G, U, l)
  case T1 ∧ T2 ⇒ (lookup(G, T1, l), lookup(G, T2, l)) match
    case (D1, D2) ⇒ intersect(D1, D2)
    case (D1, None) ⇒ D1
    case (None, D2) ⇒ D2
    case (None, None) ⇒ None
  case T1 ∨ T2 ⇒ (lookup(G, T1, l), lookup(G, T2, l)) match
    case (D1, D2) ⇒ union(D1, D2)
    case (_, _) ⇒ None
}
```

Figure 4. Alternative membership definition

1.8 Defining membership as a function

An alternative way of defining membership would be to define a recursive $lookup$ function taking a Γ, T and a label l, returning either a declaration D whose label is l, or None, indicating that T has no member named l.

A pseudo-code implementation of this $lookup$ function is given in Figure 4. Note that it's not a total function, because in some cases, it does not terminate: For instance, suppose $(x : \{L : \bot..x.L\}) \in \Gamma$. Then, $lookup(\Gamma, x.L, l)$ has to look up l in the upper bound of $x.L$, which is again $x.L$, so

Term Typing $\boxed{\Gamma \vdash t : T}$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T \textbf{ wf}}{\Gamma \vdash x : T} \text{ (TY-VAR)} \qquad \frac{\Gamma, x : T \vdash \{\overline{d}\} : T \quad \Gamma, x : T \vdash u : U \quad \Gamma \vdash U \textbf{ wf}}{\Gamma \vdash \textbf{val } x = \textbf{new } \{\overline{d}\};\ u : U} \text{ (TY-NEW)}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash u : U \quad \Gamma \vdash T \ni (m : U \to V) \quad \Gamma \vdash V \textbf{ wf}}{\Gamma \vdash t.m(u) : V} \text{ (TY-CALL)} \qquad \frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash T_1 <: T_2}{\Gamma \vdash t : T_2} \text{ (TY-SBSM)}$$

$$\frac{\Gamma, x : U, y : S \vdash t : T \quad y \notin \text{fv}(t) \quad y \notin \text{fv}(T)}{\Gamma, x : \exists (y : S) U \vdash t : T} \text{ (TY-SKOLEM)}$$

Initialization Typing $\boxed{\Gamma \vdash d : D}$

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash (L = T) : (L : T..T)} \text{ (TY-TDEF)} \qquad \frac{\Gamma \vdash T \textbf{ wf} \quad \Gamma \vdash U \textbf{ wf} \quad \Gamma, x : T \vdash u : U}{\Gamma \vdash (m(x : T) : U = u) : (m : T \to U)} \text{ (TY-MDEF)}$$

Initialization List Typing $\boxed{\Gamma \vdash \{\overline{d}\} : T}$

$$\Gamma \vdash \{\} : \top \text{ (TY-NIL)} \qquad \frac{label(d) \notin labels(\overline{d}) \quad \Gamma \vdash \{\overline{d}\} : T \quad \Gamma \vdash d : D}{\Gamma \vdash \{\overline{d}, d\} : T \wedge \{D\}} \text{ (TY-CONS)}$$

Figure 2. The exDOT Calculus: Type assignment rules

it would again call $lookup(\Gamma, x.L, l)$, leading to an infinite recursion.

To do proofs by induction, it's simpler to reason about inductively defined proof trees than to reason about invocations of a possibly non-terminating function, so that's why we prefer to define membership in terms of the two judgments given in Figure 3 rather than in terms of the $lookup$ function given in Figure 4.

But writing down the $lookup$ function reveals an important property about member lookup. It's more apparent in the $lookup$ function, but also applies to the judgment-based membership definition. It's the following: In order to look up a member l in a selection type $x.L$, we first have to look up L in the type of x, which will give a result of the form $(L : S..U)$. Then, we have to pass the result of that recursive call to another recursive call, which looks up l in U.

Reasoning about termination of recursive functions which pass the result of a recursive call to another recursive call is much harder than for simple recursive functions. A well-known such function is the McCarthy 91 function [2], and it has posed many challenges to computer-aided verification.

In previous versions of the DOT calculus (the μDOT calculus in particular), where the lemmas SUBTYP-TO-MEMBERWISE (section 3.5) and NARROW-$\ni$ (section 3.6.3) mutually depended on each other, this "double recursion" in the membership judgment was a big issue, because SUBTYP-TO-MEMBERWISE had to be applied to the conclusion returned by an inductictive hypothesis, and it was not clear whether the size of that conclusion had decreased.

We will see in section 3.6 that this problem of the $lookup$ function is not an issue any more in gDOT, because there's no such mutual dependency as in μDOT, but it's unclear whether the same problem will appear again in a type safety proof of exDOT.

1.9 Subtyping

The subtyping judgment $\Gamma \vdash S <: T$ is defined in Figure 5.

It depends on the auxiliary judgment $\Gamma \vdash D' <: D$, which compares declarations. Note that since lower bounds and method arguments are contravariant positions, the direction of the subtyping check is inverted for them.

The subtyping judgment features an explicit transitivity rule $<:$-TRANS. This saves us from proving transitivity, and

Declaration intersection $\boxed{intersect(D_1, D_2) = D_3}$

$$\frac{D_1 = (L : S_1..U_1) \qquad D_2 = (L : S_2..U_2)}{intersect(D_1, D_2) = (L : S_1 \vee S_2 \,..\, U_1 \wedge U_2)} \qquad \frac{D_1 = (m : S_1 \to U_1) \qquad D_2 = (m : S_2 \to U_2)}{intersect(D_1, D_2) = (m : S_1 \vee S_2 \;\to\; U_1 \wedge U_2)}$$

Declaration union $\boxed{union(D_1, D_2) = D_3}$

$$\frac{D_1 = (L : S_1..U_1) \qquad D_2 = (L : S_2..U_2)}{union(D_1, D_2) = (L : S_1 \wedge S_2 \,..\, U_1 \vee U_2)} \qquad \frac{D_1 = (m : S_1 \to U_1) \qquad D_2 = (m : S_2 \to U_2)}{union(D_1, D_2) = (m : S_1 \wedge S_2 \;\to\; U_1 \vee U_2)}$$

Membership $\boxed{\Gamma \vdash T \ni D}$

$$\Gamma \vdash \bot \ni (L : \top..\bot) \quad (\bot\text{-}\ni\text{-TYP}) \qquad \frac{\Gamma \vdash T_2 \ni D \quad \Gamma \vdash T_1 \not\ni label(D)}{\Gamma \vdash T_1 \wedge T_2 \ni D} \quad (\wedge\text{-}\ni\text{-2})$$

$$\Gamma \vdash \bot \ni (m : \top \to \bot) \quad (\bot\text{-}\ni\text{-MTD})$$

$$\Gamma \vdash \{D\} \ni D \quad (\text{RCD-}\ni)$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T \ni (L : S..U) \quad \Gamma \vdash U \ni D}{\Gamma \vdash x.L \ni D} \quad (\text{SEL-}\ni) \qquad \frac{\Gamma \vdash T_1 \ni D_1 \quad \Gamma \vdash T_2 \ni D_2}{\Gamma \vdash T_1 \wedge T_2 \ni intersect(D_1, D_2)} \quad (\wedge\text{-}\ni\text{-12})$$

$$\frac{\Gamma \vdash T_1 \ni D \quad \Gamma \vdash T_2 \not\ni label(D)}{\Gamma \vdash T_1 \wedge T_2 \ni D} \quad (\wedge\text{-}\ni\text{-1}) \qquad \frac{\Gamma \vdash T_1 \ni D_1 \quad \Gamma \vdash T_2 \ni D_2}{\Gamma \vdash T_1 \vee T_2 \ni union(D_1, D_2)} \quad (\vee\text{-}\ni)$$

Non-membership $\boxed{\Gamma \vdash T \not\ni l}$

$$\Gamma \vdash \top \not\ni l \quad (\top\text{-}\not\ni) \qquad \frac{\Gamma \vdash T_1 \ni D \quad \Gamma \vdash T_2 \not\ni label(D)}{\Gamma \vdash T_1 \vee T_2 \not\ni label(D)} \quad (\vee\text{-}\not\ni\text{-1})$$

$$\frac{l \neq label(D)}{\Gamma \vdash \{D\} \not\ni l} \quad (\text{RCD-}\not\ni)$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T \ni (L : S..U) \quad \Gamma \vdash U \not\ni l}{\Gamma \vdash x.L \not\ni l} \quad (\text{SEL-}\not\ni) \qquad \frac{\Gamma \vdash T_2 \ni D \quad \Gamma \vdash T_1 \not\ni label(D)}{\Gamma \vdash T_1 \vee T_2 \not\ni label(D)} \quad (\vee\text{-}\not\ni\text{-2})$$

$$\frac{\Gamma \vdash T_1 \not\ni l \quad \Gamma \vdash T_2 \not\ni l}{\Gamma \vdash T_1 \wedge T_2 \not\ni l} \quad (\wedge\text{-}\not\ni) \qquad \frac{\Gamma \vdash T_1 \not\ni l \quad \Gamma \vdash T_2 \not\ni l}{\Gamma \vdash T_1 \vee T_2 \not\ni l} \quad (\vee\text{-}\not\ni\text{-12})$$

Figure 3. The exDOT Calculus: Declaration intersection/union and membership rules

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

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash T <: T} \quad (<:\text{-REFL})$$

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash T <: \top} \quad (<:\text{-}\top)$$

$$\frac{\Gamma \vdash T \textbf{ wf}}{\Gamma \vdash \bot <: T} \quad (\bot\text{-}<:)$$

$$\frac{\Gamma \vdash D_1 <: D_2}{\Gamma \vdash \{D_1\} <: \{D_2\}} \quad (<:\text{-RCD})$$

$$\frac{\begin{array}{c}\Gamma \vdash x.L \textbf{ wf} \\ (x : T) \in \Gamma \\ \Gamma \vdash T \ni (L : S..U)\end{array}}{\Gamma \vdash x.L <: U} \quad (<:\text{-SEL-L})$$

$$\frac{\begin{array}{c}\Gamma \vdash x.L \textbf{ wf} \\ (x : T) \in \Gamma \\ \Gamma \vdash T \ni (L : S..U)\end{array}}{\Gamma \vdash S <: x.L} \quad (<:\text{-SEL-R})$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T \textbf{ wf}}{\Gamma \vdash x.\textbf{type} <: T} \quad (<:\text{-SELF-L})$$

$$\frac{\Gamma, x : S \vdash T <: U \quad \Gamma \vdash U \textbf{ wf} \quad \Gamma, x : S \vdash S \textbf{ wf}}{\Gamma \vdash \exists (x : S) T <: U} \quad (<:\text{-}\exists\text{-L})$$

$$\frac{\Gamma \vdash T_1 <: U_1 \quad \Gamma \vdash T <: U_2}{\Gamma \vdash T <: U_1 \wedge U_2} \quad (<:\text{-}\wedge)$$

$$\frac{\Gamma \vdash T_1 \textbf{ wf} \quad \Gamma \vdash T_2 \textbf{ wf}}{\Gamma \vdash T_1 \wedge T_2 <: T_1} \quad (<:\text{-}\wedge\text{-1})$$

$$\frac{\Gamma \vdash T_1 \textbf{ wf} \quad \Gamma \vdash T_2 \textbf{ wf}}{\Gamma \vdash T_1 \wedge T_2 <: T_2} \quad (<:\text{-}\wedge\text{-2})$$

$$\frac{\Gamma \vdash T_1 <: U \quad \Gamma \vdash T_2 <: U}{\Gamma \vdash T_1 \vee T_2 <: U} \quad (<:\text{-}\vee)$$

$$\frac{\Gamma \vdash T_1 \textbf{ wf} \quad \Gamma \vdash T_2 \textbf{ wf}}{\Gamma \vdash T_1 <: T_1 \vee T_2} \quad (<:\text{-}\vee\text{-1})$$

$$\frac{\Gamma \vdash T_1 \textbf{ wf} \quad \Gamma \vdash T_2 \textbf{ wf}}{\Gamma \vdash T_2 <: T_1 \vee T_2} \quad (<:\text{-}\vee\text{-2})$$

$$\frac{\Gamma \vdash T_1 <: T_2 \quad \Gamma \vdash T_2 <: T_3}{\Gamma \vdash T_1 <: T_3} \quad (<:\text{-TRANS})$$

$$\frac{(x : y.\textbf{type}) \in \Gamma \quad \Gamma \vdash x.\textbf{type wf} \quad \Gamma \vdash y.\textbf{type wf}}{\Gamma \vdash y.\textbf{type} <: x.\textbf{type}} \quad (<:\text{-SELF-R})$$

$$\frac{(x : S') \in \Gamma \quad \Gamma \vdash S' <: S \quad \Gamma \vdash T <: U}{\Gamma \vdash T <: \exists (x : S) U} \quad (<:\text{-}\exists\text{-R})$$

Declaration subtyping $\boxed{\Gamma \vdash D <: D'}$

$$\frac{\Gamma \vdash S' <: S \quad \Gamma \vdash T <: T'}{\Gamma \vdash (L : S..T) <: (L : S'..T')} \quad (\text{SUBDEC-TYP})$$

$$\frac{\Gamma \vdash S' <: S \quad \Gamma \vdash T <: T'}{\Gamma \vdash (m : S \to T) <: (m : S' \to T')} \quad (\text{SUBDEC-MTD})$$

Figure 5. The exDOT Calculus: Subtyping

it allows for simpler <:-SEL-L, <:-SEL-R, <:-∧-1, <:-∧-2, <:-∨-1 and <:-∨-2 rules, because they don't need any "baked in" transitivity (cf. the subtyping rules in [1], which don't have an explicit transitivity rule).

The <:-∃-L rule states that in order to prove that a type T depending on some object x of type S is a subtype of U if we can prove it by supposing in the environment that there is such an object x of type S. By requiring $\Gamma \vdash U$ **wf**, we ensure that the variable x does not occur in U.

The <:-∃-R rule can be seen as the introduction rule for existential types, because there is no introduction form for existential types in the grammar for terms. The rule allows us to abstract over the actual x on which T depends, by packing x into an existential.

The subtyping judgment is regular with respect to well-formedness in the same way as the term typing judgment. This means that whenever we have $\Gamma \vdash S <: T$, we know that $\Gamma \vdash S$ **wf** and $\Gamma \vdash T$ **wf**.

1.10 Well-formedness

The the well-formedness judgments $\Gamma; \overline{W} \vdash T$ **wf** and $\Gamma; \overline{W} \vdash D$ **wf** are presented in Figure 6. $\overline{W}$ is a set of types for which we already assume that they are well-formed, and if this set is empty, we simply write $\Gamma \vdash T$ **wf** (or $\Gamma \vdash D$ **wf**, respectively).

The purpose of the well-formedness judgment $\Gamma \vdash T$ **wf** is twofold: First, it ensures that all variables occuring in T are bound in Γ. Second, it rules out illegal forms of recursive types by requiring infinite derivation trees for them (which are not considered as valid derivations).

This is important because in the soundness proof, we often want to look up a member in a type, and we need to be sure that doing so terminates. But this is not always the case: For instance, suppose $(x : \{L : \bot..x.L\}) \in \Gamma$, and that we want to know if the type $x.L$ has a member named l. According to the SEL-∋ rule, we have to look up l in the upper bound of $x.L$, which is again $x.L$, so this process would not terminate.

That's why the WF-SEL rule requires U to be **wf**. If we wanted to prove $\Gamma \vdash x.L$ **wf** in the above example, we would have to apply WF-SEL, but it would require us to prove well-formedness of the bounds of $x.L$, so we would have to prove $\Gamma \vdash x.L$ **wf** again, so there is no finite derivation of $\Gamma \vdash x.L$ **wf**.

Thanks to this well-formedness judgment, we will be able to prove termination of member lookup in well-formed types by induction on the well-formedness derivation.

But checking well-formedness of the bounds in WF-SEL turns out to be too restrictive: It rules out all recursive types! For instance, suppose we have an object called $library$ of type

$$\{List : \bot..\{head : Unit \to \top\} \\ \wedge \{tail : Unit \to library.List\}\}$$

To check that $library.List$ is **wf**, we have to check that its upper bound is **wf**, which again contains $library.List$, so we would need an infinite derivation, which is not what we want.

That's why we parameterize the **wf** judgment by a set $\overline{W}$ of types for which we suppose that they are **wf**, and whenever we enter a record type, we add the record type to this set (see rule WF-RCD). We also add the rule WF-HYP, which states that for all types in $\overline{W}$, we can assume that they are **wf**. This breaks the cycle we had in the list example, and allows us to prove that $library.List$ is **wf**.

2. Examples

2.1 Recursion in gDOT

The following example defines an infinite stream of $unit$ objects. Note that it can entirely be defined in gDOT, without needing any features from exDOT.

$$\begin{array}{l}\textbf{val } glob = \textbf{ new } \{ \\ \quad E = \top \\ \quad Stream = \{head : \top \to glob.E\} \\ \qquad \wedge \{tail : \top \to glob.Stream\} \\ \}; \\ \textbf{val } unit = \textbf{ new } \{\}; \\ \textbf{val } stream = \textbf{ new } \{ \\ \quad head(x : \top) : glob.E = unit \\ \quad tail(x : \top) : glob.Stream = stream \\ \}; \\ stream.tail(unit).tail(unit).head(unit)\end{array}$$

2.2 Encoding refinement types

Refinement types as in [1] can be encoded using existentials: The refinement type $T\,\{z \Rightarrow D_1, \ldots D_n\}$ can be represented as $\exists(z : T \wedge \{D_1\} \wedge \ldots \wedge \{D_n\})\ z.\textbf{type}$.

2.3 Higher-kinded types

The schema proposed in [3] to encode higher-kinded types depends on Scala's # operator, which selects a type member on a type, because applications of type lambdas will have to select the member called `Apply` on the type lambda, as for example in

```
(Lambda1 { type Apply = List[$hkArg$0] }
   { type $hkArg$0 = B }) # Apply
```

With the existentials presented here, we can represent type selections $C\#T$ as $\exists(x : C)x.T$.

3. Type safety

This section presents a type safety proof for gDOT. The detailed proof was developed in Coq and can be found on GitHub. [1]

[1] https://github.com/samuelgruetter/dot-calculus/blob/master/stable/stable_typ-cbmode-precise-gDot.v

Well-formed types $\boxed{\Gamma; \overline{W} \vdash T \textbf{ wf}}$

$$\Gamma; \overline{W} \vdash \top \textbf{ wf} \quad \text{(WF-}\top\text{)}$$

$$\Gamma; \overline{W} \vdash \bot \textbf{ wf} \quad \text{(WF-}\bot\text{)}$$

$$\Gamma; \overline{W} \vdash W_i \textbf{ wf} \quad \text{(WF-HYP)}$$

$$\frac{\Gamma; \overline{W}, \{D\} \vdash D \textbf{ wf}}{\Gamma; \overline{W} \vdash \{D\} \textbf{ wf}} \quad \text{(WF-RCD)}$$

$$\frac{(x : T) \in \Gamma \quad \Gamma; \overline{W} \vdash T \textbf{ wf}}{\Gamma; \overline{W} \vdash x.\textbf{type wf}} \quad \text{(WF-SELF)}$$

$$\frac{(x : T) \in \Gamma \quad \Gamma \vdash T \ni (L : S..U) \quad \Gamma; \overline{W} \vdash T \textbf{ wf}, S \textbf{ wf}, U \textbf{ wf}}{\Gamma; \overline{W} \vdash x.L \textbf{ wf}} \quad \text{(WF-SEL)}$$

$$\frac{\Gamma; \overline{W} \vdash T_1 \textbf{ wf}, T_2 \textbf{ wf}}{\Gamma; \overline{W} \vdash T_1 \wedge T_2 \textbf{ wf}} \quad \text{(WF-AND)}$$

$$\frac{\Gamma; \overline{W} \vdash T_1 \textbf{ wf}, T_2 \textbf{ wf}}{\Gamma; \overline{W} \vdash T_1 \vee T_2 \textbf{ wf}} \quad \text{(WF-OR)}$$

$$\frac{\Gamma, x : T; \overline{W} \vdash T \textbf{ wf}, U \textbf{ wf}}{\Gamma; \overline{W} \vdash \exists (x : T) U \textbf{ wf}} \quad \text{(WF-}\exists\text{)}$$

Well-formed declarations $\boxed{\Gamma; \overline{W} \vdash D \textbf{ wf}}$

$$\frac{\Gamma; \overline{W} \vdash S \textbf{ wf}, U \textbf{ wf}}{\Gamma; \overline{W} \vdash L : S..U \textbf{ wf}} \quad \text{(WF-TMEM)}$$

$$\frac{\Gamma; \overline{W} \vdash S \textbf{ wf}, U \textbf{ wf}}{\Gamma; \overline{W} \vdash m : S \to U \textbf{ wf}} \quad \text{(WF-MTD)}$$

Figure 6. The exDOT Calculus: Well-Formedness

3.1 Stable types

The proof we present here further restricts gDOT by requiring that in selection types $x.L$, the type of x is a *stable* type.

Definition: *The judgment "T* **stable***" is defined by the following inference rules:*

$$\top \textbf{ stable} \qquad \frac{T_1 \textbf{ stable} \quad T_2 \textbf{ stable}}{T_1 \wedge T_2 \textbf{ stable}}$$

$$\bot \textbf{ stable}$$

$$\{D\} \textbf{ stable} \qquad \frac{T_1 \textbf{ stable} \quad T_2 \textbf{ stable}}{T_1 \vee T_2 \textbf{ stable}}$$

It excludes selection types at top level (i.e. behind $\wedge$ and $\vee$), but it allows them inside record types.

As we will see, stable types have the property that their set of members is preserved by narrowing, or, in other words, this set remains stable, so that's why we call these types stable.

3.2 Inverting well-formedness judgments

Throughout the proof, we will often have to invert well-formedness judgments. These inversion lemmas are trivial, except for the case WF-RCD, because this rule grows the set of assumptions $\overline{W}$ in its hypothesis, but we always need **wf** judgments with an empty $\overline{W}$. So we have to prove the following lemma:

Lemma INVERT-WF-RCD:
If $\Gamma \vdash \{D\}$ **wf***, then* $\Gamma \vdash D$ **wf**.
Proof: By inversion, we get $\Gamma; \{D\} \vdash D$ **wf**. Now we can apply REMOVE-HYP-FROM-WF with $\overline{V}$ empty and $\overline{W}$ the singleton set containing only the record type $\{D\}$. □

Lemma REMOVE-HYP-FROM-WF:
If $\Gamma; \overline{V} \vdash U$ **wf** *and* $\Gamma; \overline{W} \vdash D$ **wf**,
then $\Gamma; (\overline{V} \cup (\overline{W} - U)) \vdash D$ **wf**.
The $-$ sign denotes removal of one element from a set.
Proof: We simultaneously prove the same statement for well-formed types, by mutual induction over the well-formedness derivations. In the case WF-HYP when $W_i = U$, i.e. W_i is the type we are removing from the assumptions $\overline{W}$, we cannot use WF-HYP any more, but we have to plug in the $\Gamma; \overline{V} \vdash U$ **wf** derivation that this lemma took as first hypothesis. □

3.3 Weakening

The cases RED-CALL-1 and RED-CALL-2 of the preservation theorem will need weakening lemmas stating that if a derivation holds in some typing environment, it also holds in an extended environment.

For example, the weakening lemma for the membership judgment is the following:

Lemma WEAKEN-$\ni$: *If* $\Gamma \vdash T \ni D$*, then* $\Gamma, \Gamma' \vdash T \ni D$.

We prove such weakening lemmas for all judgments, by (mutual) induction on the derivations.

3.4 Properties of the membership judgment

Looking up a member in a well-formed type is a total function:

Lemma $\ni$-TOTAL-FUNC: *Given a* Γ*,* T *and* l*, if* $\Gamma \vdash T$ **wf***, exactly one of the following two statements holds:*

- *There exists a unique* D *whose label is* l*, such that* $\Gamma \vdash T \ni D$
- $\Gamma \vdash T \not\ni l$

Proof: By induction on the **wf** derivation. □

It's important to notice that this only holds if the **wf** judgment is strong enough, because we need it as "fuel" for the induction. In particular, it has to recursively check well-formedness of the upper bounds of selection types, but without ruling out recursive types. Section 1.10 explains how this is achieved.

Another lemma which looks simple at first sight, but only holds because of a powerful **wf** judgment, is the following:

Lemma $\ni$-PRESERVES-WF: *If* $\Gamma \vdash T \ni D$ *and* $\Gamma \vdash T$ **wf***, then* $\Gamma \vdash D$ **wf**.
Proof: By induction on the membership judgment. □

These two lemmas will be needed by the SUBTYP-TO-MEMBERWISE lemma presented in the next section.

3.5 Memberwise subtyping

Definition: *Let us define memberwise subtyping, written* $\Gamma \vdash T_1 <:_{memberwise} T_2$*, as follows:*

$$\forall D_2,\ \Gamma \vdash T_2 \ni D_2 \Rightarrow \exists D_1,\ \Gamma \vdash T_1 \ni D_1\ \wedge\ \Gamma \vdash D_1 <: D_2$$

Note that memberwise subtyping does not imply ordinary subtyping: For instance, if the types $x.L_1$ and $x.L_2$ are both lower-bounded by $\bot$ and upper-bounded by $\top$, we have a counterexample. This is desired, because otherwise there would be no nominality.

The other direction however (almost) holds: Ordinary subtyping implies memberwise subtyping. Proving it by induction on the subtyping derivation works in all cases except in the case <:-SEL-R: We are given a subtyping statement of the form $\Gamma \vdash S <: x.L$, and $\Gamma \vdash x.L \ni D_2$, and we have to find a D_1 such that $\Gamma \vdash S \ni D_1$ and $\Gamma \vdash D_1 <: D_2$. But by inverting the subtyping judgment, we don't get any useful hypotesis. In particular, we don't have any inductive hypothesis, because the <:-SEL-R rule has no subtyping hypothesis. This problem can easily be fixed by adding $\Gamma \vdash S <: U$ to the <:-SEL-R rule:

$$\frac{\Gamma \vdash x.L \textbf{ wf} \qquad (x : T) \in \Gamma \qquad \Gamma \vdash T \ni (L : S..U) \qquad \Gamma \vdash S <: U}{\Gamma \vdash S <: x.L}\ (<:\text{-SEL-R}')$$

Note that there's no need to do the same for the <:-SEL-L rule, because the members of a selection type are the same as the members of its upper bound.

So why don't we always use this <:-SEL-R′ rule? The reason is that narrowing for subtyping would not hold any more. Section 3.6.2 explains why.

So let us define the the following variant of subtyping:

Definition: *The* $\Gamma \vdash T_1 <:_{checkbounds} T_2$ *judgment is defined by the same rules as the ordinary subtyping judgment, with the only difference that the* <:-SEL-R *rule is replaced by the above* <:-SEL-R-′ *rule.*

Now we can state the lemma:

Lemma SUBTYP-TO-MEMBERWISE:
If $\Gamma \vdash T_1 <:_{checkbounds} T_2$*,*
then $\Gamma \vdash T_1 <:_{memberwise} T_2$.
Proof: By induction on the subtyping derivation. The cases <:-REFL and <:-$\bot$ need $\ni$-PRESERVES-WF because the sub-declaration judgment to be proved needs well-formedness of the declarations.

In the case <:-$\wedge$-1, the given subtyping judgment is of the form $\Gamma \vdash U_1 \wedge U_2 <:_{checkbounds} U_1$. The type U_1 contains a given declaration D_2, and we now have to check if U_2 also has a declaration with the same label, in which case the declaration becomes more precise, or if it doesn't, in which case the declaration remains the same. To do so, we need the $\ni$-TOTAL-FUNC lemma. And to prove the sub-declaration statements, we will also need $\ni$-PRESERVES-WF.

Of course, the same applies symmetrically for the case <:-$\wedge$-2. □

So we can now turn ordinary subtyping into memberwise subtyping, but we still have to show how to turn $\Gamma \vdash T_1 <: T_2$ into $\Gamma \vdash T_1 <:_{checkbounds} T_2$. We will do so in section 3.9.

3.6 Narrowing

In the case RED-CALL of the preservation proof, we will have a term of the form $x.m(y)$ stepping to $[y/z_i]u_i$, where z_i is the name of the formal parameter of the method m, and u_i is the body of m. The argument type of m will be some type U, but due to subsumption, the actual precise type of y in Γ might be a $U' <: U$. Originally, the method body u_i was typechecked assuming that the method argument has type U, but now we know that it has the more precise type U'. That's why we need a narrowing lemma, stating that if a typing derivation holds in some environment, it also holds in a more precise environment.

By "more precise environment", we mean a sub-environment according to the following definition:

Definition: *We call Γ' a sub-environment of Γ, and write it as $\Gamma' <: \Gamma$, if Γ' can be written as $\Gamma_1, x : S', \Gamma_2$, and Γ can be written as $\Gamma_1, x : S, \Gamma_2$, and $\Gamma' \vdash S' <:_{checkbounds} S$.*

Note that we require $checkbounds$ because we will have to apply SUBTYP-TO-MEMBERWISE on this subtyping derivation.

3.6.1 Narrowing term typing derivations

The narrowing lemma for terms is the following:

Lemma NARROW-TY-TRM:
If $\Gamma \vdash t : T$ and $\Gamma' <: \Gamma$, then $\Gamma' \vdash t : T$.
Proof: By mutual induction on term typing, initialization typing, and initialization list typing. Since term typing has subsumption (TY-SBSM), we can simply insert a subsumption step at all occurrences of the variable whose type is narrowed.

In the case TY-CALL, we have to narrow a membership judgment (which does not have subsumption), so we need NARROW-$\ni$, which will be presented later in this section.

And in order to narrow the case TY-SBSM itself, we have to narrow a subtyping judgment. □

3.6.2 Narrowing subtyping derivations

So we need the following lemma:

Lemma NARROW-SUBTYPE:
If $\Gamma \vdash T_1 <: T_2$ and $\Gamma' <: \Gamma$, then $\Gamma' \vdash T_1 <: T_2$.

Note that in subtyping and in the judgments on which subtyping depends, there is no subsumption rule (see section 4.1 for the reasons), so we cannot prove it in the same simple way as NARROW-TY-TRM.

Before proving the lemma, let us see why it doesn't hold in $checkbounds$ mode: Suppose we have an Int and $String$ type, none of which is a subtype of the other, and that $(x : \{L : Int..\top\}) \in \Gamma$, which is narrowed to $(x : \{L : Int..\top\} \wedge \{L : \bot..String\}) \in \Gamma'$.

$\Gamma \vdash Int <: x.L$ can be proven using <:-SEL-R′, but $\Gamma' \vdash Int <: x.L$ doesn't hold, because the type of x now has the declaration

$$intersect((L : Int..\top), (L : \bot..String)) = (L : (Int \vee \bot)..(\top \wedge String))$$

So in order to apply <:-SEL-R′, we would have to prove $\Gamma' \vdash (Int \vee \bot) <: (\top \wedge String)$, which requires proving $\Gamma' \vdash Int <: String$, which doesn't hold.

That's why we don't use <:-SEL-R′, but <:-SEL-R instead.

Besides this, proving NARROW-SUBTYPE can be done easily by applying SUBTYP-TO-MEMBERWISE on the subtyping derivation contained in $\Gamma' <: \Gamma$, and by using the two lemmas NARROW-$\ni$ and NARROW-WF presented next.

3.6.3 Narrowing membership derivations

Lemma NARROW-HAS: *If $\Gamma \vdash T \ni D_2$ and $\Gamma' <: \Gamma$ and $\Gamma' \vdash T$* **wf***, there exists a D_1 such that $\Gamma' \vdash T \ni D_1$ and $\Gamma' \vdash D_1 <: D_2$.*

Note that the well-formedness of T is checked in Γ', not in Γ. This is quite a strong hypothesis, but we need it because we have to prove a $\Gamma' \vdash D_1 <: D_2$ statement, which requires well-formedness of D_1 and D_2 in Γ'.

Proof: By mutual induction on membership and non-membership derivations. In the cases <:-∧-1 and <:-∧-2, we have to apply $\ni$-PRESERVES-WF on the $\Gamma' \vdash T$ **wf** derivation in order to get the well-formedness needed to prove $\Gamma' \vdash D_1 <: D_2$. □

3.6.4 Narrowing well-formedness derivations

Now we still have to prove NARROW-WF. We only need it with an empty set of assumptions $\overline{W}$, but in order to make the induction work, we also have to prove it for non-empty $\overline{W}$:

Lemma NARROW-WF:
If $\Gamma; \overline{W} \vdash T$ **wf** *and $\Gamma' <: \Gamma$, then $\Gamma'; \overline{W} \vdash T$* **wf***.*
Proof: By mutual induction on the **wf** derivations for types and declarations.

The only interesting case is WF-SEL: Since we have the restriction that selection types $x.L$ are only well-formed if the type of x is stable, we know that the bounds of $x.L$ do not change under narrowing, and proving this case is easy. □

However, if we lift the stable type restriction, it becomes much harder, and no succint explanation of why it's so hard has been found yet. This question is investigated in a separate Coq file on GitHub. [2]

3.7 The substitution lemmas

In the case RED-CALL of the preservation proof, where a term of the form $x.m(y)$ steps to $[y/z_i]u_i$, with z_i being the name of the formal parameter of the method m, and u_i being the body of m, the term u_i was typechecked in the environment $\Gamma, z_i : T$, but after the reduction step, we have to give a typing derivation of $[y/z_i]u_i$ in Γ, which doesn't have any binding for z_i.

This is achieved using the substitution lemma:

Lemma SUBST-TY:
If $\Gamma, x : S \vdash t : T$ and $(y : [y/x]S) \in \Gamma$,
then $[y/x]\Gamma \vdash [y/x]t : [y/x]T$.

Note that we only have to substitute variables for variables, because the reduction rule RED-CALL only is applicable when the argument is fully evaluated, i.e. when it is a variable bound in the store.

[2] https://github.com/samuelgruetter/dot-calculus/blob/master/dev/existential/why-stable_typ-cbmode-precise-gDot.v

We also need similar substitution lemmas for all the other judgments, but they all look the same and are thus omitted here.

3.8 Good bounds

Definition: *We say that a type T has good bounds in the enviroment Γ if*
$\forall\, L\ S\ U,\ \Gamma \vdash T \ni (L : S..U) \Rightarrow \Gamma \vdash S <:_{checkbounds} U$.
Moreover, we extend this definition to environments and say that Γ has good bounds if all types bound by Γ have good bounds.

A disturbing property of most sufficiently powerful versions of DOT is that given a Γ and a $\Gamma' <: \Gamma$, both having good bounds, and a type T having good bounds in Γ, it might happen that T does not have good bounds in Γ' any more. Or, in other words, the good bounds property is not preserved by narrowing, even if both environments have good bounds.

The following counterexample proves this claim: Let's again use Int and $String$, none of which is a subtype of the other, and let $\Gamma = \{(x : \{L : Int..\top\}), (y : \{L : \bot..\top\})\}$ and $T = x.L \wedge y.L$. Note that Γ has good bounds, and the only member of T is $(L : (Int \vee \bot)..(\top \wedge \top))$, so T has good bounds as well.

Let $\Gamma' = \{(x : \{L : Int..\top\}), (y : \{L : \bot..String\})\}$, which also has good bounds, but T does not have good bounds in Γ' any more, for the same reason as in the counterexample in section 3.6.2.

All DOT type safety proofs have to carefully circumnavigate the need for this property, which makes them considerably more delicate.

3.9 Subtyping with checked bounds

In order to use the narrowing lemmas, we have to give a $\Gamma' <: \Gamma$ derivation, which requires subtyping in $checkbounds$ mode, because it uses SUBTYP-TO-MEMBERWISE, and direct invocations of SUBTYP-TO-MEMBERWISE (will occur in the case TY-CALL of the progress proof) also require subtyping in $checkbounds$ mode.

So we need the following lemma:

Lemma SUBTYP-TO-CHECKBOUNDS: *If $\Gamma \vdash T_1 <: T_2$ and Γ has good bounds, then $\Gamma \vdash T_1 <:_{checkbounds} T_2$.*
Proof: Straightforward mutual induction on subtyping and sub-declaration derivations. □

Note that if we want to extend this proof to exDOT, where the rule <:-∃-L extends the environment in its premise, we will have to extend the good bounds judgment as well, which might turn out to be tricky.

3.10 Typechecking the store

In order to state the soundness results, we need a judgment to typecheck the store, for which we will prove that it's preserved by reduction.

Definition: *Let $\emptyset_\Gamma$ denote the empty typechecking environment, and let $\emptyset_s$ denote the empty store. We define store typing, written as $s \models \Gamma$, by the following two rules:*

$$\emptyset_s \models \emptyset_\Gamma \qquad \frac{s \models \Gamma \quad \Gamma, x : T \vdash \{\overline{d}\} : T \quad x \notin dom(s) \quad x \notin dom(\Gamma)}{s, x \mapsto \{\overline{d}\} \models \Gamma, x : T}$$

3.11 Soundness results

We finally have all helper lemmas to prove the standard PROGRESS and PRESERVATION theorems:

Theorem PROGRESS: *If $\Gamma \vdash t : T$ and $s \models \Gamma$, either there exist a term t' and a store s' such that $t \mid s \rightarrow t' \mid s'$, or t is a result, i.e. a variable bound in s.*
Proof: By induction on the typing derivation. The interesting case is when t is of the form $x.m(y)$: We have $\Gamma \vdash x : S, \Gamma \vdash y : U$, and $\Gamma \vdash S \ni (m : U \to V)$, but because of subsumption, the precise type of x might be a subtype S' of S. So to find that S' also has a method member named m, we have to use the lemma SUBTYP-TO-MEMBERWISE (see section 3.5). But before we can apply it, we need to convert the $\Gamma \vdash S' <: S$ derivation to $checkbounds$ mode using SUBTYP-TO-CHECKBOUNDS. This requires that Γ has good bounds, but since $s \models \Gamma$, we know that the type members of all types in Γ have their lower bound equal to their upper bound, so good bounds for Γ trivially holds by <:-REFL. □

Theorem PRESERVATION: *If $s \models \Gamma$ and $\Gamma \vdash t : T$ and $t \mid s \rightarrow t' \mid s'$, there exists a Γ' such that $s' \models \Gamma, \Gamma'$ and $\Gamma, \Gamma' \vdash t' : T$.*
Proof: First, note that Γ' will always be an environment with at most one binding, because each reduction step adds at most one new object to the store.

The proof is by induction on the reduction derivation. The only interesting case is the case RED-CALL, where we have a term of the form $x.m(y)$ stepping to $[y/z_i]u_i$, where z_i is the name of the formal parameter of the method m, and u_i is the body of m. The argument type of m has some type U, but due to subsumption, the actual precise type of y in Γ might be a $U' <: U$. Originally, the method body u_i was typechecked assuming that the method argument has type U, but now we know that it has the more precise type U'. So we have to apply NARROW-TY to show that the body also typechecks in the environment $\Gamma, z_i : U'$. And then, we can apply SUBST-TY to show that $[y/z_i]u_i$ typechecks in Γ. □

4. Discussion

4.1 Precise and imprecise typing

We call typing rules which include a subsumption rule *imprecise*, and we call those which don't *precise*.

4.1.1 On the term level

The type assignment rules of exDOT are imprecise. One could also make them precise, and add explicit subsumption to the TY-CALL rule (that's what [1] did).

But this is problematic because of the TY-NEW rule: It requires that in **val** $x = \textbf{new}\ \{\overline{d}\};\ u$, the type of u does not contain x (because x is not bound in Γ), which restricts the expressivity of the calculus if typing is precise. With imprecise typing, however, we also must keep this restriction, but we can apply subsumption while typechecking u to make x disappear.

To fix this, one could add explicit subsumption to TY-NEW as well, but it turns out that the whole calculus and in particular the formulation of the preservation theorem and its proof become much simpler with imprecise term typing.

4.1.2 On the type level

All the other rules of exDOT are precise, even though it seems at first sight that using imprecise typing would be much simpler:

First, we would not need a membership judgment any more, and the many occurrences of the pattern consisting of $(x : T) \in \Gamma$ and $\Gamma \vdash T \ni D$ could simply be replaced by $\Gamma \vdash x : \{D\}$.

Second, the narrowing proof, which is the most delicate part in the gDOT soundness proof, would become trivial: When narrowing the type of x from S to a subtype S', it would be sufficient to add a subsumption step from S' to S at all occurrences of x.

But when trying out many different proof strategies for an imprecise calculus, it turned out that we always need a judgment saying that Γ has good bounds. Defining this judgment without precise typing seems to be impossible, because bad bounds can always be turned into good bounds by subsumption. For instance, if the precise type of x is $\{L : Int..\top\} \wedge \{L : \bot..String\}$, these bad bounds can simply be subsumed to $\{L : \bot..\top\}$, which are good.

4.2 What should well-formedness test?

There are various conditions on types that a well-formedness judgment could test. We could summarize them as follows:

1. For all $x.L$ occuring inside the type, x is bound in the environment.
2. The process of repeatedly following the upper bound of a selection type until we only have record types, intersections and unions, always terminates. Or in other words: Looking up members in a type terminates.
3. All $(L : S..U)$ declarations occuring inside the type are such that $S <: U$.
4. The type does not contain contradicting $(L : S_1..U_1)$ and $(L : S_2..U_2)$, i.e. $S_1 <: U_2$ and $S_2 <: U_1$.

It is important to be aware of the difference between 3 and 4.

In the calculus presented here, well-formedness checks 1 and 2, because this seems to be strong enough, but it does not check for 3 and 4, which has the advantage that well-formedness does not depend on subtyping.

For 3 and 4, we have used the notion of "good bounds", so it's not the case that these conditions are unimportant, but it's not a good idea to include them into the more low-level well-formedness judgment, because as we have seen in section 3.8, good bounds are not preserved by narrowing.

4.3 Membership in existential types

One might be surprised that there is no membership rule with a conclusion of the form $\Gamma \vdash \exists(x : T)U \ni \dots$. The reasons for this are first that instead of looking up a member in an existential, we can apply the TY-SKOLEM rule before to get rid of the existential, and then do the lookup with the existing membership rules, and second that it is hard to define membership for existentials in a satisfying way.

Let us review two attempts: We could add the following rule:

$$\frac{\Gamma, x : T \vdash U \ni D \quad x \notin fv(D)}{\Gamma \vdash \exists(x : T)U \ni D}$$

But it is not satisfying because of the $x \notin fv(D)$ condition, which makes lookup in existentials non-total.

So we could try to get rid of this condition by quantifying x in the returned declaration. We would first define a helper function:

$$\begin{array}{l} distribute(x, T, D) := D\ \textbf{match}\ \{ \\ \quad \textbf{case}\ (L : S..U) \Rightarrow (L : (\exists(x : T)S)..(\exists(x : T)U)) \\ \quad \textbf{case}\ (m : S \to U) \Rightarrow (m : (\exists(x : T)S) \to (\exists(x : T)U)) \\ \} \end{array}$$

And then, we could add the rule

$$\frac{\Gamma, x : T \vdash U \ni D}{\Gamma \vdash \exists(x : T)U \ni distribute(x, T, D)}$$

However, this is not satisfying either, because the existential is duplicated, so we lose the information that the existential in the argument type of the method declaration refers to the same x as the existential in the return type.

The problem that we cannot find a satisfying membership rule for existentials is relevant because there are situations when the current TY-SKOLEM rule is not sufficient to do the lookups in the existentials.

For instance, suppose

$$\Gamma = \{(x : \{L : \bot..\exists(z : T)U\}), (a : x.L)\}$$

and that we want to do skolemization before typechecking $a.m(\dots)$ so that we don't have to look up the member m inside an existential. The current TY-SKOLEM rule only applies if the precise type of a in Γ is an existential, but here it's a subtype of an existential.

Various ways of making the TY-SKOLEM rule more powerful have been tried, but non of them was satisfactory.

5. Future work

Future work includes lifting the stable type restriction (section 3.1), defining a more powerful TY-SKOLEM rule to overcome the problems described in section 4.3, and finally to prove type safety of the full exDOT calculus.

References

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

[2] D. E. Knuth. Artificial intelligence and mathematical theory of computation. chapter Textbook Examples of Recursion, pages 207–229. Academic Press Professional, Inc., San Diego, CA, USA, 1991. ISBN 0-12-450010-2. URL `http://arxiv.org/abs/cs/9301113`.

[3] M. Odersky. Higher-kinded types in dotty v2, 2014. URL `https://github.com/lampepfl/dotty/blob/bdb6361911/docs/HigherKinded-v2.md`.