Submitted to OOPSLA 2015

Type-Dependent Name Resolution

Hendrik van Antwerpen
Delft University of Technology
h.vanantwerpen@student.tudelft.nl

Pierre Neron
Delft University of Technology
p.j.m.neron@tudelft.nl

Andrew Tolmach
Portland State University
apt@cs.pdx.edu

Eelco Visser
Delft University of Technology
visser@acm.org

Guido Wachsmuth
Delft University of Technology
guwac@acm.org

Abstract

We describe a language-independent theory for name and type analysis.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.3.2 [*Programming Languages*]: Language classifications; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; D.3.4 [*Programming Languages*]: Processors; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages; D.2.6 [*Software Engineering*]: Programming Environments

Keywords Language Specification; Name Binding; Types; Domain Specific Languages; Meta-Theory

1. Introduction

Name resolution and type resolution are two fundamental concerns in programming language specification and implementation. Name resolution means determining the identifier declaration corresponding to each identifier use in a program. Type resolution means determining the type of each identifier and expression in the program, as part of performing type checking or inference. These two tasks are essential components of many language processing tools, including interpreters, compilers and IDEs. Moreover, precise descriptions of name and type resolution are essential parts of a formal language semantics. [Mention language workbench??] Yet there are as yet no universally accepted formalisms that support both specification and implementation of these tasks. This is in notable contrast to the situation with syntax definition, for which context-free grammars provide a well-established declarative formalism that underpins a wide variety of useful tools.

In this paper, we show how two existing formalisms, scope graphs and type constraints, can be extended and combined to fill this gap. Our formalisms: (i) have a clear and clean underlying theory; (ii) handle a broad range of common language features; (iii) are declarative, but are realizable by practical algorithms and tools; (iv) are factored into language-specific and language-independent parts, to maximize re-use; and (v) apply to erroneous programs (for which resolution fails or is ambiguous) as well as to correct ones. Moreover, although name and type resolution are obviously related, as far as possible we treat them as separate concerns; this improves modularity and helps clarify exactly what the relationships between these two tasks are.

Our starting point is recent work by Neron *et al.* [2], which shows how name resolution for lexically-scoped languages can be formalized in a way that meets the criteria above. The name binding structure of a program is captured in a *scope graph* which records identifier declarations and references and their scoping relationships, while abstracting away program details. Its building blocks are *scopes*, which are minimal programs regions that behave uniformly with respect to resolution. Each scope can contain identifier declarations and references, each tagged with its position in the original AST. Scopes can be connected by a *parent* relation modelling lexical nesting, and sometimes by a named *import* relation to model access to named modules, classes, and similar structures. A scope graph is constructed from the program AST using a language-dependent traversal, but thereafter, it can be processed in a language-independent way. A *resolution calculus* gives a formal definition of what it means for a reference to identifier x at position i to resolve to a declaration of x at position j, written $x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}$. A given reference may to resolve to PN ▶*to resolve to ?*◀ one, none, or many declarations. There is a sound and complete *resolu-*

```
fun y1:Int {
  fun y2:Bool {
    let x3:Bool = if y4 then True else y5
    in x6
  }
}
```

Figure 1. A simple program

tion algorithm that computes the set of declarations to which each reference resolves.

As an example, consider the program in Figure 1 in an expression-oriented language with standard lexical scoping rules. Subscripts on identifiers represent AST positions. From the scope graph for this program, the resolution algorithm will generate the facts

$$y_4^{\mathsf{R}} \longmapsto y_2^{\mathsf{D}}, y_5^{\mathsf{R}} \longmapsto y_2^{\mathsf{D}}, x_6^{\mathsf{R}} \longmapsto x_3^{\mathsf{D}}$$

Note that resolution takes account of the shadowing of the binding at $\mathtt{x}_1$ by the binding at $\mathtt{x}_2$ PN ►*isn't it y instead of x ?*◄; this is the whole point of the resolution calculus.

Scope graphs do not include explicit type information. However, if the language associates types with identifier declarations, it is easy to obtain the type of an identifier reference by first resolving the reference to a declaration and then looking up the associated type information by position in the AST.[1] For example, the program of Figure 1 carries this associated type information:

$$\mathtt{TypeAt}(1) = \mathtt{Int}, \mathtt{TypeAt}(2) = \mathtt{Bool}, \mathtt{TypeAt}(3) = \mathtt{Bool}$$

This lookup facility can be made available as a service to a separate type resolution mechanism.

One well-known mechanism for type resolution, which meets our formalism criteria above, is based on extracting *constraints* on types and type variables from the AST and then using *unification* to solve the constraints and instantiate the variables. This approach can be used for type checking or (more importantly) type inference; under suitable language restrictions, it infers principal, i.e. most general, types. This technique goes back at least to Milner's seminal paper on polymorphism [1], and has since been extended to cover many additional language features, notably subtyping; Pottier and Remy (Chapter 10 of [3]) give a detailed exposition, and show how an efficient resolution algorithm can be expressed using rewrite rules.

Figure 2 illustrates how a constraint-based approach to type checking might work on top of name resolution for a small fragment of a simple monomorphically-typed language. At the top of the figure, we show the set of constraints, obtained by walking over the program AST; we defer discussion of the details of constraint generation to a later section. There are two forms of constraints: conventional type equalities, and name resolution constraints. The latter replace the implicit name resolution via manipulation of a typing context that would be used in a more conventional presentation. Note that constraints can mention *position variables* ι as well as ordinary type variables τ. At the bottom of the figure, we show a solution to the constraint set, written as a substiution on type and position variables. The solution is obtained by plugging in the facts about $\longmapsto$ and $\mathtt{TypeAt}()$ derived from the scope graph, and then performing unification; again, we defer details to a later section.

Constraints (where overall program type is τ_0)

$$\begin{array}{rcl} \tau_0 & = & \mathtt{Int} \to \tau_1 \\ \tau_1 & = & \mathtt{Bool} \to \tau_2 \\ y_4^{\mathsf{R}} & \longmapsto & y_{\iota_1}^{\mathsf{D}} \\ \mathtt{TypeAt}(\iota_1) & = & \mathtt{Bool} \\ y_5^{\mathsf{R}} & \longmapsto & y_{\iota_2}^{\mathsf{D}} \\ \mathtt{TypeAt}(\iota_2) & = & \mathtt{Bool} \\ x_6^{\mathsf{R}} & \longmapsto & x_{\iota_3}^{\mathsf{D}} \\ \mathtt{TypeAt}(\iota_3) & = & \tau_2 \end{array}$$

Solution to constraints

$\iota_1 = 2, \iota_2 = 2, \iota_3 = 3, \tau_2 = \mathtt{Bool}$,
$\tau_1 = \mathtt{Bool} \to \mathtt{Bool}, \tau_0 = \mathtt{Int} \to \mathtt{Bool} \to \mathtt{Bool}$

Figure 2. Constraints for Simple Program

```
record A1 {x2:Int}
record B3 {a4:A5, x6:Bool}
...
y7.x8 // what is the type of y7 ?
y9.a10.x11 // what are the types of y9, y9.a10 ?
```

Figure 3. Program with records

This simple two stage approach—name resolution using the scope graph followed by separate type resolution—will work for many language constructs. But the full story is more complicated, because sometimes name resolution also depends on type resolution. Consider the program fragments in Figure 3, written in a language having nominal records and using standard dot notation for record field access. In a scope graph setting, we can create one scope that contains the field declaration of record type $\mathtt{A}_1$, and another that contains those of $\mathtt{B}_3$, and associate each scope with the corresponding type name. Now, in order to resolve the type of $\mathtt{y}_7\mathtt{.x}_8$ we must first resolve the field name $\mathtt{x}_8$ to the appropriate declaration field ($\mathtt{x}_2$ or $\mathtt{x}_6$). But this name resolution depends on the *type* of $\mathtt{y}_7$, so we must resolve that type first. In general, we may need arbitrarily deep recursion between the two kinds of resolution. For example, to handle the nested record dereference on the last line, we must first resolve the name of $\mathtt{y}_9$, then its type, then the name of $\mathtt{a}_{10}$, then *its* type, and finally the name and type of $\mathtt{x}_{11}$. PN ►*Why emphasize one "its"?*◄

To solve this challenge, we reformulate the task of generating a scope graph from a given program as one of finding

[1] There is nothing special about types in this regard: the same mechanism can be used to obtain other declaration attributes such as record field offsets, visibility attributes, etc.

a minimal solution to a set of *scope constraints* obtained by an AST traversal. Scope constraints are analogous to typing constraints, but are resolved using a different (and simpler) algorithm. We then introduce a class of *scope variables* and modify the resolution calculus to characterize resolution in potentially *incomplete* scope graphs (i.e., graphs characterized by constraints involving unresolved scope variables). We can then interleave (partial) scope graph resolution and type unification until a complete instantiation of all variables (types, positions, and scopes) is obtained. This approach permits us to resolve all the names and types for the record examples of Figure 3 and for a broad range of other language constructs.

Our specific contributions are as follows:

- We define a constraint language that can express both typing and name binding constraints, parameterized by an underlying notion of type compatibility, and define satisfiability for problems in this language.
- We show how to extend the name resolution calculus and algorithm of [2] to handle incomplete scope graphs.
- We describe details of constraint generation for a simple toy language that illustrates many of the interesting resolution issues associated with modules, classes, and records.
- We describe an algorithm for solving problems in our constraint language instantiated to use nominal subtyping, and show that it is sound (and complete?????? PN ►*Unlikely for the generic constraint language since variables can be everywhere, maybe if we restrict variable to where they actually appear in our collection*◄).

Our constraint generator and solver have been implemented and will be submitted as artifacts to accompany the paper.

[Roadmap may follow or be incorporated in above.]

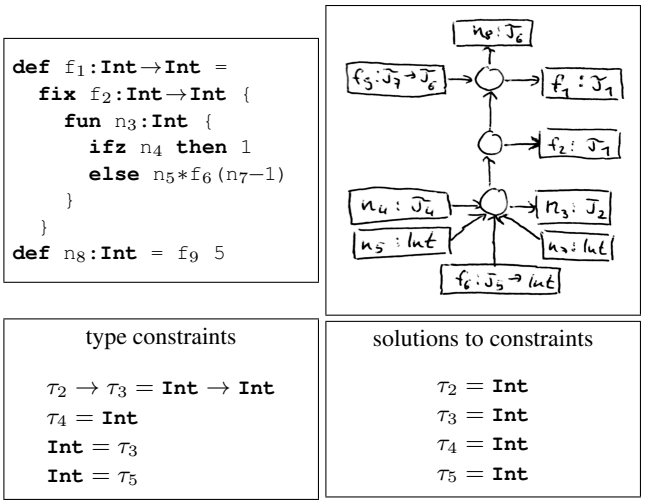

Figure 4. Type checking.

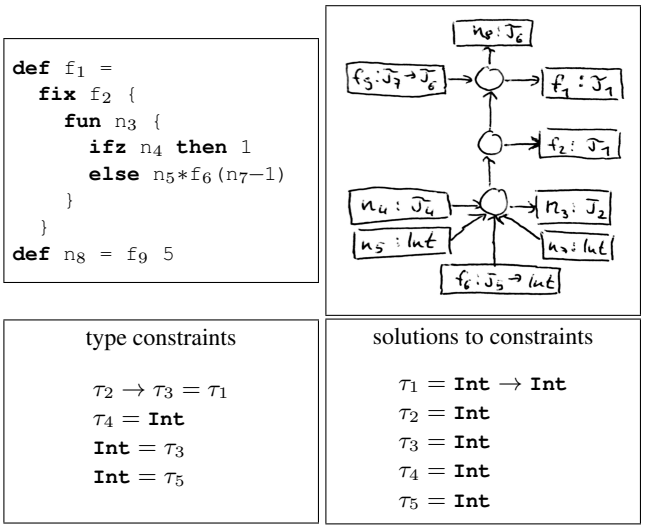

Figure 5. Type inference.

2. Scope Graphs + Types (Guido)

separation of name resolution and typing vanilla type constraints LM with type inference

example scope graphs + types for interesting examples can we draw those? or do we use constraints? but those are not so easy to read

consider example programs + scope graphs + type assignments, i.e. the result of resolution

discuss what the solutions for type-dependent problems look like

2.1 Scope Graphs

2.2 Resolution

2.3 Type Checking

2.4 Type Inference

2.5 Nominal Types

2.6 Type-Dependent Resolution

2.7 Nominal Subtyping

3. Extended scope graphs

Unlike previous version where first build graph then resolve, here we collect all these fact in one traversal

3.1 Scope Graphs

The notion of *scope graph* has been introduced in [2], it is a generic model of the name binding structure of a program. This model is built around 3 basic notions derived from the program AST:

- definitions are occurrences of variable names that introduce names, x_i^{D} denotes the definition of name x at position i in the program, the position i can be omitted for simplicity
- references are occurrences of variable names referring to already defined name, a reference with name x at position i is denoted x_i^{R} and the position i can be omitted for simplicity
- scopes are abstractions over a set of nodes that behaves uniformly with respect to name binding, a scope is usually denoted S_j or s_j with j an integer.

In order to form the graph, we define several relations between these different elements. Every scope S can have at most one *parent* scope denoted $\mathcal{P}(S)$, it also has a set of associated declarations $\mathcal{D}(S)$ and one of associated references $\mathcal{R}(S)$. Each declaration or references belongs to a unique scope. These relations form the most basic scope graph structure and already allows us to represent usual binding structure like different flavors of let bindings or function arguments bindings.

In order to handle more complex name binding patterns and in particular modules and imports, two more relations are added to the graph:

- A scope has an associated set of references called imports, denoted $\mathcal{I}(S)$. These references refers to module names that are imported in the scope S, for example with a statement like `import x` or `include x`.
- A declaration can have an associated scope. For example the definition of a new module `module A {...}` is a declaration of the name `A` with an associated scope S_1 where S_1 is the scope of the body of the module definition. Such a declaration is denoted $\mathtt{A}_i^{\mathsf{D}}{:}S_1$.

Given this model, a resolution calculus has been presented in [2]. This resolution calculus defines a resolution relation between the references and the declaration of a graph. This resolution aims at building a path leading from a reference to a declaration, it is done in a context of $\mathbb{I}$ of seen imports ($\mathbb{I}$ is a set of references).

A path p is a list of steps (i.e. scope transitions) representing the scope transitions in the graph. A step is either a parent step $\mathbf{P}$, an import step $\mathbf{I}(y^{\mathsf{R}}, y^{\mathsf{D}}{:}S)$ where y^{R} is the imports used and $y^{\mathsf{D}}{:}S$ its corresponding declaration or a declaration step $\mathbf{D}(x^{\mathsf{D}})$ leading to a declaration x^{D}. Given a seen import

Edges in scope graph

$$\frac{\mathcal{P}(S_1) = S_2}{\mathbb{I} \vdash \mathbf{P} : S_1 \longrightarrow S_2} \quad (P)$$

$$\frac{y_i^{\mathsf{R}} \in \mathcal{I}(S_1) \setminus \mathbb{I} \quad \mathbb{I} \vdash p : y_i^{\mathsf{R}} \longmapsto y_j^{\mathsf{D}}{:}S_2}{\mathbb{I} \vdash \mathbf{I}(y_i^{\mathsf{R}}, y_j^{\mathsf{D}}{:}S_2) : S_1 \longrightarrow S_2} \quad (I)$$

$$\frac{S_2 \in \mathcal{IS}(S_1)}{\mathbb{I} \vdash \mathbf{I}(S_2) : S_1 \longrightarrow S_2} \quad (S)$$

Transitive closure

$$\frac{}{\mathbb{I} \vdash [] : A \twoheadrightarrow A} \quad (N)$$

$$\frac{\mathbb{I} \vdash s : A \longrightarrow B \quad \mathbb{I} \vdash p : B \twoheadrightarrow C}{\mathbb{I} \vdash s \cdot p : A \twoheadrightarrow C} \quad (T)$$

Reachable declarations

$$\frac{x_i^{\mathsf{D}} \in \mathcal{D}(S') \quad \mathbb{I} \vdash p : S \twoheadrightarrow S' \quad WF(p)}{\mathbb{I} \vdash p \cdot \mathbf{D}(x_i^{\mathsf{D}}) : S \rightarrowtail x_i^{\mathsf{D}}} \quad (R)$$

Visible declarations

$$\frac{\mathbb{I} \vdash p : S \rightarrowtail x_i^{\mathsf{D}} \quad \forall j, p'(\mathbb{I} \vdash p' : S \rightarrowtail x_j^{\mathsf{D}} \Rightarrow \neg(p' < p))}{\mathbb{I} \vdash p : S \longmapsto x_i^{\mathsf{D}}} \quad (V)$$

Reference resolution

$$\frac{x_i^{\mathsf{R}} \in \mathcal{R}(S) \quad \{x_i^{\mathsf{R}}\} \cup \mathbb{I} \vdash p : S \longmapsto x_j^{\mathsf{D}}}{\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_j^{\mathsf{D}}} \quad (X)$$

Figure 6. Resolution calculus

set $\mathbb{I}$, a path p is a valid resolution in the graph from reference x_i^{R} to declaration x_i^{D} when the following statement holds:

$$\mathcal{I}(\vdash)p : x_i^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$$

This resolution relation is parametrized by two predicates on paths, a well-formedness predicate $WF(p)$ and an ordering relation $<$ that allows us to encode different name binding policies, e.g. transitive or non transitive imports. One usual definition of the well formedness predicate is called *no parents after imports*, meaning that the path have to be of the form $\mathbf{P}^* \cdot \mathbf{I}(\_,\_)^* \cdot \mathbf{D}(\_)$ and a usual order is the lexicographic ordering extending $\mathbf{D}(\_) < \mathbf{I}(\_,\_) < \mathbf{P}$

Figure 6 presents the rules defining the resolution relation as presented [2] with an extra rule D that will be introduced in the following section.

3.2 Direct Imports

The scope graphs as introduced in [2] already offer a large coverage of different binding constructors that can be found in existing programming languages. However it does not allow interaction between name binding and typing, i.e. when the resolution of a reference to its definition depends on the type of another expression. For example, in an OO language, the resolution of a.x has to lead to the definition of x that appear in the class definition (i.e. the type) A of object a.

TODO ► ***Another example is Pascal's*** with ***constructor (as introduced in ...)*** ◄

In order to model these behaviors we extend the scope graph with a new kind of imports called direct import TODO ►***(or anonymous import ? scope import ?)***◄ that imports a scope into another scope without the use of any reference. In addition to its set of associated imports (that are references of the form x^{R}), a scope now also has a set of direct imports $\mathcal{IS}(S)$ that are other scopes of the graph. With these imports we introduce the corresponding scope transition rule similar to the (I) rule of the original calculus:

$$\frac{S_2 \in \mathcal{IS}(S_1)}{\mathbb{I} \vdash \mathbf{I}(S_2) : S_1 \longrightarrow S_2} \quad (D)$$

The complete resolution calculus with this new rule is presented in Figure 6.

4. Constraint Language

Name resolution as presented in [2] relies on a two stage approach, first the language dependent construction of a scope graph from the program AST and then the resolution of all the references in this program to their corresponding declarations. With explicit typing, typing rules can simply use the resolution relation to recover the type of a variable reference by using the type that is declared at the variable declaration site. This makes the type resolution depend on the name resolution. However, for some language constructors, the most common example being field access in records or objects, the resolution of a name to its declaration depends on the type of another expression: in a field access, a.x, in order to resolve x, one first needs to find the type of a and then to look for x in that type definition. This scheme induces a dependency, not only of the name resolution but also of the scope graph construction (one does not know in which scope the reference x lies) on the type resolution. The constraint approach allows us to resolve this mutual dependency by generating constrained not only related to the type resolution but also to the scope graph construction and the name resolution.

4.1 Syntax

The resolution of a program first involves a collection of the constraints relative to this program using an AST traversal. Constraints generated by this traversal are not only related to typing but also to the scope graph construction and the name resolution. Therefore these constraints relate not only types but also scopes, references, declarations and positions of the program AST. All these elements can also be represented using variables of the corresponding sort in the constraint language (that we usually denote with greek letters). The language independent base terms of the constraint language are:

- Scopes in $\langle S \rangle$ which are either ground scopes denoted s_i or variables σ
- Declarations in $\langle D \rangle$ which are either ground declarations x_i^{D} or variables δ
- References in $\langle R \rangle$ which are either ground references x_i^{R} or variables ρ
- Positions in $\langle I \rangle$ which can be ground positions i, positions of a declaration or a reference $Pos(d)/Pos(r)$ or variables ι

The types $\langle T \rangle$ are the only base terms of the constraint language that are not language independent. Given $C_{\mathcal{T}}$ the set of type constructors of the constraint language, the types can be constructed with these constructors and with type variables τ.

Assumptions Given these terms we can now define the syntax of constraints. We distinguish two categories of constraints, the first one called assumption, defined by the sort $\langle A \rangle$ corresponds to known facts about the program. They

$$
\begin{array}{lllll}
\langle A \rangle := & P(\langle S \rangle, \langle S \rangle) & \mid \langle S \rangle \in \mathcal{IS}(\langle S \rangle) & \mid \langle T \rangle <: \langle T \rangle \\
\mid & \langle R \rangle \in \mathcal{R}(\langle S \rangle) & \mid \langle R \rangle \in \mathcal{I}(\langle S \rangle) & \mid \langle T \rangle \rightsquigarrow \langle S \rangle \\
\mid & \langle D \rangle \in \mathcal{D}(\langle S \rangle) & & \\
\\
\langle C \rangle := & \mathsf{True} & \mid \langle R \rangle \mapsto \langle D \rangle & \mid \kappa =_\kappa \kappa \\
\mid & \langle C \rangle \wedge \langle C \rangle & \mid \langle T \rangle \rightsquigarrow \langle S \rangle & \mid \langle T \rangle \leq \langle T \rangle \\
\mid & \langle I \rangle : \langle T \rangle & &
\end{array}
$$

$$
\begin{array}{llllll}
\langle S \rangle := & \sigma \mid s_i & \langle I \rangle := & \iota \mid i \mid \langle D \rangle^{\mathsf{P}} \mid \langle R \rangle^{\mathsf{P}} \\
\langle R \rangle := & \rho \mid x^{\mathsf{R}}_{\langle I \rangle} & \langle T \rangle := & \tau \mid C_{\mathcal{T}}(\kappa, ..., \kappa) \\
\langle D \rangle := & \delta \mid x^{\mathsf{D}}_{\langle I \rangle} & &
\end{array}
$$

Figure 7. Syntax of constraints

mainly corresponds to the scope graph construction but also allows us to extend the subtyping relation ($<:$) and to define the scope associated with a type definition ($\rightsquigarrow$). Note that these assumption can contain variables.

Then $\langle C \rangle$ defines the language of real constraints, this set include the constant True, conjunction $\wedge$, resolution constraints $\longmapsto$, type to scope association constraints $\rightsquigarrow$, sort equality constraints $=_\kappa$ between two terms of sort κ.

TODO ▶*Remark types can contain definitions ...*◀
TODO ▶

- ***We want type to be uniquely defined (mapping type to scopes for example), thus in a nominal type system, the nominal types are not identified by their names but by their declaration so we can distinguish between two different types with the same name (in different part of the program).***
- ***Do we need to represent references and declarations as*** x_i ***APT*** ▶**yes, and should add this to LHS of reference resolution constraint or is just the position** i **enough? Including the names will allow (gramatically) for the reference and the scope to have different names, in which case we'll need extra conditions elsewhere to ensure they're the same**◀.
- ***We need to indicate uniqueness/multiplicity in the type resolution constraints. Which options:*** 1, $+$***? Or only annotate if you allow multiple?*** ***APT*** ▶**Don't understand this point.**◀

◀

4.2 Semantics of the constraints

Assumptions The role of the assumptions is to provide the model of the program in which the constraints have to be valid. Given a set of assumptions A, we denote $A^{<:}$ the set of assumptions of the form $T_1 <: T_2$ in A, $A^{\rightsquigarrow}$ the set of assumptions of the form $T \rightsquigarrow S$ in A and $A^{\mathcal{G}}$ the subset formed by the other assumptions. The interpretation of a ground set of assumptions defines three different artifacts, the scope graph of the program, the sub-typing relation and a partial mapping from types to scopes.

Scope graph The scope graph interpretation of A corresponds to the set $A^{\mathcal{G}}$ of assumptions of the form $\langle R \rangle \in$

$\mathcal{R}(\langle S\rangle)$, $\langle D\rangle \in \mathcal{D}(\langle S\rangle)$, $\langle R\rangle \in \mathcal{I}(\langle S\rangle)$, $\langle S\rangle \in \mathcal{IS}(\langle S\rangle)$, $P(\langle S\rangle, \langle S\rangle)$, assuming it corresponds to the definition of a correct scope graph as introduced in section 3 (a declaration or a reference belongs to a unique scope, each scope has at most one parent). In this case this scope graph is denoted $\mathcal{G}_A$.

Type to scopes The type to scope mapping interpretation of A is given by the set $A^{\rightsquigarrow}$ of assumptions of the form $T \rightsquigarrow S$, assuming this set defines a mapping from scopes to types (i.e. a type has at most one associated types). This mapping is then denoted Σ_A.

Sub-typing The subtyping interpretation of A is not as straightforward since it is language dependent. Therefore for a language we require the interpretation of a set S of assumptions of the form $T_1 <: T_2$ into a subtyping relation to be provided, this subtyping relation is denoted $\preceq_S$ and has to respect the following properties:

- for all S, $\preceq_S$ is reflexive and transitive
- the function $S \to \preceq_S$ is monotonic (if $S_1 \subseteq S_2$ then $\preceq_{S_1} \subseteq \preceq_{S_2}$)
- if $T_1 <: T_2 \in S$ then $T_1 \preceq_S T_2$

For example, in the LMR language, the sub-typing relation $\preceq_S$ associated with the set of sub-typing assumption S is defined by the following rules:

$$\frac{}{T \preceq_S T} \qquad \frac{T_1 \preceq_S T_2 \quad T_2 \preceq_S T_3}{T_1 \preceq_S T_3}$$

$$\frac{T_1 <: T_2 \in S}{T_1 \preceq_S T_2} \qquad \frac{T_1 \preceq_S S_1 \quad S_2 \preceq_S T_2}{S_1 \to S_2 \preceq_S T_1 \to T_2}$$

By extension the subtyping relation associated to a set of assumptions A is denoted $\preceq_A$. PN ►*Should we enforce the sub typing relation to be well-founded ?*◄

Given a ground set of assumption A, we denote $[\![A]\!]$ the interpretation of A as the corresponding triple $\mathcal{G}_A, \Sigma_A, \preceq_A$

4.3 Interpretation of constraints

We also define the interpretation of a ground set of constraints. This interpretation is done in a context containing several elements:

- A scope graph $\mathcal{G}$
- A sub-typing relation $\preceq$ between types
- A type to scope mapping Σ
- A typing environment mapping positions to types

A context $\mathcal{G}, \Sigma, \preceq, \psi$ satisfies a constraint C if the predicate $\mathcal{G}, \Sigma, \preceq, \psi \models C$ holds. This predicate is defined by the following set of rules, where $=$ is the syntactic equality on terms modulo reduction of the position projection on declarations and references (i.e. $Pos(x_i^{\mathsf{R}}) \to i$):

$$\frac{}{\mathcal{G}, \Sigma, \preceq, \psi \models \mathsf{True}} \quad \text{(C-TRUE)}$$

$$\frac{\mathcal{G}, \Sigma, \preceq, \psi \models C_1 \quad \mathcal{G}, \Sigma, \preceq, \psi \models C_2}{\mathcal{G}, \Sigma, \preceq, \psi \models C_1 \wedge C_2} \quad \text{(C-AND)}$$

$$\frac{\psi(i) = T}{\mathcal{G}, \Sigma, \preceq, \psi \models i : T} \quad \text{(C-TYPEOF)}$$

$$\frac{\Sigma(T) = S}{\mathcal{G}, \Sigma, \preceq, \psi \models T \rightsquigarrow S} \quad \text{(C-SCOPEOF)}$$

$$\frac{\vdash_{\mathcal{G}} p : x^R_{i_1} \mapsto x^D_{i_2}}{\mathcal{G}, \Sigma, \preceq, \psi \models i_1 \mapsto i_2} \quad \text{(C-RESOLVE)}$$

$$\frac{t_1 = t_2}{\mathcal{G}, \Sigma, \preceq, \psi \models t_1 =_\kappa t_2} \quad \text{(C-EQ)}$$

$$\frac{T_1 \preceq T_2}{\mathcal{G}, \Sigma, \preceq, \psi \models T_1 \leq T_2} \quad \text{(C-SUBTYPE)}$$

Note that unlike the set of assumptions which has to be ground to be interpreted, a constraint does not necessarily have to be ground to be satisfiable. This allows us to type the identity function as $X \to X$ where X is a free type variable.

4.4 Program resolution

The collection of a program constraint generate a set of fact that mixes assumptions and constraints but these elements will be treated separately. Given a program p, we denote A_p be set of assumption produced by the collection and C_p the leftover constraint. Both A_p and C_p contain free variables, moreover, some of these variables are even shared between constraints and assumptions. In order to resolve a program p, one need to build a multi-sorted substitution ϕ and a typing environment ψ such that:

$$[\![\phi(A_p)]\!], \psi \models \phi(C_p) \qquad (\diamond)$$

Where $\phi(X)$ denote the application of the substitution ϕ to all the variable appearing in X that are in the domain of ϕ. Note that ϕ has to make $\phi(A_p)$ a set of ground assumptions whereas some free variable may remain in $\phi(C_p)$. When the proposition $\diamond$ holds we say that ψ and ϕ resolve p.

PN ►*Next paragraph is dubious, there is probably substyping involved*◄

Principal solution Allowing free variable in a solution allows us to state what the principal resolution of a program might be, that is a solution that allow to recover all the other possible resolutions. That is, a pair ϕ, ψ is said to be the principal resolution of a program p if it satisfies $\diamond$ and for all ϕ', ψ' resolving p then ϕ' and ψ' are instances of ϕ and ψ, i.e.

$$\exists \phi'' \; s.t. \; \phi' = \phi'' \cdot \phi \wedge \forall i, \phi''(\psi(i)) = \psi'(i)$$

PN ►*We probably need to relax this, saying that, for types we only have* $\phi(\tau) \preceq \phi''(\phi(\tau))$ *and* $=$ *for other find of variables, same for* $\phi''(\psi(i)) \preceq \psi'(i)$ ◄

5. Constraint Collection (Transcribe from Hendrik's code)

The collection rules for LMR are specified in DynSem as following:

```
module extraction

imports
  ds-signatures/LMR-sig
  ds-signatures/Desugared-sig
  ds-signatures/Constraints-sig
  trans/abstract-syntax/abstract-syntax
  trans/desugar/desugar

signature
  internal-sorts
    DeclScope
    Stmts
    DefBinds
    FldBinds
    RecDecls
    TyC

  internal-constructors
    Stmts : List(Stmt) -> Stmts
    DefBinds : List(DefBind) -> DefBinds
    FldBinds : List(FldBind) -> FldBinds
    RecDecls : List(Decl) -> RecDecls
    TyC : Type * C -> TyC
    DeclScope : Scope -> DeclScope

  arrows
    Program -> C
    Stmt -> C
    Stmts -> C
    Expr -> C
    DefBind -> C
    DefBinds -> C
    FldBind -> C
    FldBinds -> C
    RecDecls -> C
    Super -> C
    Decl -> C
    QualRef -> C

    TypeX -tyc-> TyC

    QualRef -id-> RefId

    DeclScope -> Scope

rules

  Program(stmts)
    -> AndStar([ FParent(s,NoneScope())
                 , c
                 ])
    where
      Scope(sfresh()) => s
    , Scope s |- Stmts(stmts) -> c

rules

  Scope s |- Module(id,stmts)
    -> AndStar([ FDecl(id,s)
                 , FAssoc(id,s')
                 , FParent(s',SomeScope(s))
                 , c
                 ])
    where
      Scope(sfresh()) => s'
    , Scope s' |- Stmts(stmts) -> c

  Scope s |- Import(qmid)
    -> AndStar([ c
                 , FImportR(mid,s)
                 ])
    where
      qmid -> lala , lala -> c
    , qmid -id-> mid

  Scope s |- Def(b) -> c
    where
      DeclScope DeclScope(s) |- b -> c

rules

  Scope s |- Rec(id,super,decls)
    -> AndStar([ FDecl(id,s)
                 , FAssoc(id,s')
                 , FParent(s',SomeScope(s))
                 , FTypeOf(POfDecl(id),TRec(id))
                 , sc
                 , fc
                 ])
    where
      Scope(sfresh()) => s'
    , DeclScope DeclScope(s') Type TRec(id) |- super -> sc
    , DeclScope DeclScope(s') |- RecDecls(decls) -> fc

  DeclScope ds Type ty |- Super(qtid)
    -> AndStar([ tc
                 , FImportR(tid,ds)
                 , CResolves(tid,d,rpos(tid))
                 , FSubType(ty,TRec(d))
                 ])
    where
      qtid -> lala , lala -> tc
    , qtid -id-> tid
    , DeclVar(sfresh()) => d

  Type ty |- NoSuper()
    -> FSubType(ty,TTop())

rules

  Type ty |- i@IntLit(_)
    -> AndStar([ FTypeOf(epos(i),ty)
                 , CSubType(TInt(),ty,epos(i))
                 ])

  Type ty |- b@BinExpr(e1,_,e2)
    -> AndStar([ FTypeOf(epos(b),ty)
                 , CSubType(TInt(),ty,epos(b))
                 , c1
                 , c2
                 ])
    where
      Type TInt() |- e1 -> c1
    , Type TInt() |- e2 -> c2

  Type ty |- i@IfzThenElse(ec,et,ef)
    -> AndStar([ FTypeOf(epos(i),ty)
                 , cc
                 , ct
                 , cf
                 ])
    where
      Type TInt() |- ec -> cc
    , et -> ct
    , ef -> cf

  Type ty |- r@Ref(qid)
    -> AndStar([ FTypeOf(epos(r),ty)
                 , qc
                 , CResolves(id,d,epos(r))
                 , CTypeOf(POfDecl(d),dty,epos(r))
                 , FTypeOf(POfRef(id),dty)
                 , CSubType(dty,ty,epos(r))
                 ])
    where
      qid -> lala , lala -> qc
    , qid -id-> id
    , DeclVar(sfresh()) => d
    , TVar(sfresh()) => dty

  Scope s Type ty |- f@Fix(a,e)
    -> AndStar([ FTypeOf(epos(f),ty)
                 , FParent(s',SomeScope(s))
                 , ca
                 , ce
                 ])
    where
      Scope(sfresh()) => s'
    , DeclScope DeclScope(s') |- a -> ca
    , Scope s' |- e -> ce

  Scope s Type ty |- f@Fun(a,e)
    -> AndStar([ FTypeOf(epos(f),ty)
                 , FParent(s',SomeScope(s))
                 , ca
                 , ce
                 , CSubType(TArrow(ty1,ty2),ty,epos(f))
                 ])
    where
      Scope(sfresh()) => s'
    , TVar(sfresh()) => ty1
    , DeclScope DeclScope(s') Type ty1 |- a -> ca
    , TVar(sfresh()) => ty2
    , Scope s' Type ty2 |- e -> ce

  Type ty |- a@App(e1,e2)
    -> AndStar([ FTypeOf(epos(a),ty)
                 , c1
                 , c2
                 ])
    where
      TVar(sfresh()) => ty1
    , Type TArrow(ty1,ty) |- e1 -> c1
    , Type ty1 |- e2 -> c2

rules

  Let([b|bs],e) -> LetPar([b],Let(bs,e))
  Let([],e) -> e

  Scope s Type ty |- l@LetRec(binds,e)
    -> AndStar([ FTypeOf(epos(l),ty)
                 , FParent(s',SomeScope(s))
                 , cbs
                 , ce])
    where
      Scope(sfresh()) => s'
    , DeclScope DeclScope(s') Scope s' |- DefBinds(binds) -> cbs
    , Scope s' |- e -> ce

  Scope s Type ty |- l@LetPar(binds,e)
    -> AndStar([ FTypeOf(epos(l),ty)
                 , FParent(s',SomeScope(s))
                 , cbs
                 , ce])
    where
```

$$\frac{S \vdash sts \longrightarrow c}{\vdash Prog(sts) \longrightarrow P(S, \bot) \wedge c}$$

$$\frac{S \vdash s \longrightarrow c_1 \quad S \vdash sts \longrightarrow c_2}{\vdash s :: sts \longrightarrow c_1 \wedge c_2}$$

$$\frac{S' \vdash sts \longrightarrow c}{S \vdash Mod(x_i, sts) \longrightarrow x_i^{\mathsf{D}}{:}S' \in \mathcal{D}(S) \wedge P(S', S) \wedge c}$$

$$\frac{\vdash qid \longrightarrow c \quad qid \xrightarrow{\mathsf{R}} id_i^{\mathsf{R}}}{S \vdash Imp(qid) \longrightarrow id_i^{\mathsf{R}} \in \mathcal{I}(S) \wedge c}$$

$$\frac{S^D \vdash b \longrightarrow c}{S \vdash Def(b) \longrightarrow c}$$

$$\frac{S^D \vdash decls \longrightarrow fc \quad S'^D\ \mathcal{R}(x_i^{\mathsf{D}})^T \vdash sup \longrightarrow sc}{S \vdash Rec(x, sup, dec)_i \longrightarrow x_i^{\mathsf{D}}{:}S' \in \mathcal{D}(S) \wedge \mathcal{R}(x_i^{\mathsf{D}}) \rightsquigarrow S' \wedge P(S', S) \wedge i : \mathcal{R}(x_i^{\mathsf{D}}) \wedge fc \wedge sc}$$

$$\frac{\vdash qtid \longrightarrow tc \quad qtid \xrightarrow{\mathsf{R}} x_i^{\mathsf{R}}}{S^D\ T^T \vdash Super(qid) \longrightarrow x_i^{\mathsf{R}} \in \mathcal{I}(S) \wedge x_i^{\mathsf{R}} \longmapsto \delta \wedge ty <: \mathcal{R}(\delta) \wedge tc}$$

$$\frac{}{T^T \vdash NoSuper \longrightarrow T <: \bot}$$ (is it necessary ?)

$$\frac{}{T^T \vdash Int(n) \longrightarrow i : T \wedge Int <: T}$$ (Why Ftypeof ?)

$$\frac{op :: T_1 * T_2 \to T3 \quad T_1^T \vdash e_1 \longrightarrow c_1 \quad T_2^T \vdash e_2 \longrightarrow c_2}{ty^T \vdash Bop(op, e_1, e_2)_i \longrightarrow i : ty \wedge T_3 \leq ty \wedge c_1 \wedge c_2}$$

$$\frac{\mathbb{B}^T \vdash e_1 \longrightarrow c_1 \quad T^T \vdash e_2 \longrightarrow c_2 \quad T^T \vdash e_3 \longrightarrow c_3}{T^T \vdash Ift(e_1, e_2, e_3)_i \longrightarrow i : T \wedge c_1 \wedge c_2 \wedge c_3}$$

$$\frac{\vdash qid \longrightarrow c \quad qid \xrightarrow{\mathsf{R}} x_{i'}^{\mathsf{R}}}{T^T \vdash Ref(qid)_i \longrightarrow i : T \wedge x_{i'}^{\mathsf{R}} \longmapsto \delta \wedge \delta^{\mathsf{P}} : \tau \wedge i' : \tau \wedge \tau \leq T \wedge c}$$

$$\frac{S'^D\ \tau^T \vdash b \longrightarrow c_b \quad S'\ \tau^T \vdash e \longrightarrow c_e}{S\ T^T \vdash Fix(b, e)_i \longrightarrow i : T \wedge P(S', S) \wedge \tau \leq T \wedge c_b \wedge c_e}$$ (This rule is wrong in DS)

$$\frac{S'^D\ \tau_1^T \vdash b \longrightarrow c_b \quad S'\ \tau_2^T \vdash e \longrightarrow c_e}{S\ T^T \vdash Fun(b, e)_i \longrightarrow i : T \wedge P(S', S) \wedge \tau_1 \to \tau_2 \leq T \wedge c_b \wedge c_e}$$

$$\frac{(\tau_1 \to T)^T \vdash e_1 \longrightarrow c_1 \quad \tau_1^T \vdash e_2 \longrightarrow c_2}{T^T \vdash App(e_1, e_2)_i \longrightarrow i : T \wedge c_1 \wedge c_2}$$

Figure 8. Constraint generation

```
      Scope(sfresh()) ⇒ s'
    , DeclScope DeclScope(s') ⊢ DefBinds(binds) → cbs
    , Scope s' ⊢ e → ce

  DefBind(decl,e)
    → AndStar([ cd
              , ce
              ])
    where
      TVar(sfresh()) ⇒ ty
    , Type ty ⊢ decl → cd
    , Type ty ⊢ e → ce

rules

  Type ty ⊢ n@New(qtid,binds)
    → AndStar([ FTypeOf(epos(n),ty)
              , tc
              , CResolves(tid,d,epos(n))
              , CTypeOf(POfDecl(d),tyd,epos(n))
              , CSubType(tyd,ty,epos(n))
              , FParent(s',NoneScope())
              , FImportR(tid,s'')
              , fc
              ])
    where
      qtid → lala , lala → tc
    , qtid −id↦ tid
    , DeclVar(sfresh()) ⇒ d
    , TVar(sfresh()) ⇒ tyd
    , Scope(sfresh()) ⇒ s'
    , DeclScope DeclScope(s') ⊢ FldBinds(binds) → fc

  DeclScope ds ⊢ FldBind(id,e)
    → AndStar([ FRef(id,ds)
              , CResolves(id,d,rpos(id))
              , CTypeOf(POfDecl(d),ty,rpos(id))
              , FTypeOf(POfRef(id),ty)
              , c
          ])
    where
      DeclVar(sfresh()) ⇒ d
    , TVar(sfresh()) ⇒ ty
    , Type ty ⊢ e → c

  Type ty ⊢ a@FldAccess(e,vid)
    → AndStar([ FTypeOf(epos(a),ty)
              , c
              , FParent(s,NoneScope())
              , FImportS(se,s)
              , FRef(id,s)
              , CAssoc(tye,se,epos(a))
              , CResolves(id,d,epos(a))
              , CTypeOf(POfDecl(d),tyd,epos(a))
              , FTypeOf(POfRef(id),tyd)
              , CSubType(tyd,ty,epos(a))
              ])
    where
      vid → id
    , TVar(sfresh()) ⇒ tye
    , Type tye ⊢ e → c
    , Scope(sfresh()) ⇒ s
    , DeclVar(sfresh()) ⇒ d
    , TVar(sfresh()) ⇒ tyd
    , ScopeVar(sfresh()) ⇒ se

  Scope s Type ty ⊢ w@With(e1,e2)
    → AndStar([ FTypeOf(epos(w),ty)
              , c1
              , FParent(s',SomeScope(s))
              , FImportS(s'',s')
              , CAssoc(ty1,s'',epos(w))
              , c2
              ])
    where
      TVar(sfresh()) ⇒ ty1
    , Type ty1 ⊢ e1 → c1
    , Scope(sfresh()) ⇒ s'
    , Scope s' ⊢ e2 → c2
    , ScopeVar(sfresh()) ⇒ s''

rules

  DeclScope sd Type ty ⊢ Decl(id)
    → AndStar([ FDecl(id,s)
              , FTypeOf(POfDecl(id),ty)
              ])
    where
      sd → s

  DeclScope sd Type ty ⊢ TypedDecl(id,tyx)
    → AndStar([ FDecl(id,s)
              , FTypeOf(POfDecl(id),dty)
              , CTypeOf(POfDecl(id),ty,dpos(id))
              , dc
              ])
    where
      sd → s
    , tyx −tyc↦ TyC(dty,dc)

rules

  TXInt() −tyc↦ TyC(TInt(),True())

  TXArrow(ty1,ty2) −tyc↦ TyC(TArrow(ty1',ty2'),AndStar([c1,c2]))
    where
      ty1 −tyc↦ TyC(ty1',c1)
    , ty2 −tyc↦ TyC(ty2',c2)

  TXRec(qid) −tyc↦ TyC(ty,c)
    where
      qid → lala , lala → qc
    , qid −id↦ id
    , DeclVar(sfresh()) ⇒ d
    , TVar(sfresh()) ⇒ ty
    , AndStar([ CResolves(id,d,rpos(id))
              , CTypeOf(POfDecl(d),ty,rpos(id))
              , qc
              ]) ⇒ c

rules

  Scope s ⊢ UnqualRef(id) → FRef(id,s)

  QualRef(qid,id)
    → AndStar([ FParent(s',NoneScope())
              , c
              , FImportR(id',s')
              , FRef(id,s')
              ])
    where
      qid → c
    , qid −id↦ id'
    , Scope(sfresh()) ⇒ s'

  UnqualRef(id) −id↦ id
  QualRef(_,id) −id↦ id

rules

  DeclScope(s) → s

  Stmts([stmt|stmts]) → AndStar([c,cs])
    where
      stmt → c
    , Stmts(stmts) → cs
  Stmts([]) → True()

  DefBinds([bind|binds]) → AndStar([c,cs])
    where
      bind → c
    , DefBinds(binds) → cs
  DefBinds([]) → True()

  FldBinds([bind|binds]) → AndStar([c,cs])
    where
      bind → c
    , FldBinds(binds) → cs
  FldBinds([]) → True()

  RecDecls([decl|decls]) → AndStar([c,cs])
    where
      Type TVar(sfresh()) ⊢ decl → c
    , RecDecls(decls) → cs

  RecDecls([]) → True()
```

6. Resolution Algorithm (Hendrik)

name resolution in incomplete scope graphs (as extension of the ESOP algorithm) solving of type constraints given some restriction of constraints what is the interaction between the two?

Along with this new feature we also allow these new import to be scope variables. Indeed, these direct imports are used to represent scope import induced by a typing relation. For example, given the expression $\mathsf{a.x}$ where a is an object, we interpret the construction $\mathsf{a.\_}$ as opening new scope (e.g. s_1) with no parent and one direct import of the scope corresponding to the type (class definition) of a, say s_a. Then x will be resolved in this scope s_1. However, when starting the resolution, one can not know what is this imported scope s_a since finding it first requires to type a which requires to resolve the reference a to its definition. Therefore when building the graph we do not know yet what is the scope imported in s_1 and model this by adding a direct import to s_1 identified by a variable σ. In the set of constraint, this variable σ is bound to the type of the expression a and will, at some point, be resolved to the type of a allowing us to resolve x in s_1 using the import s_a.

A scope graph containing variables is called an incomplete scope graph.

6.1 Resolution in an incomplete scope graph

We have now introduced a cyclic dependency between typing and scope graph construction. Typing of an expression obviously depends on the scope graph since it requires resolution of references appearing in the expression, but the scope graph (i.e. the direct imports) also depends on the typing of some expressions in the program. However, it is still possible to resolve some references, even if the scope graph is incomplete. For example in the expression $\mathsf{y} + \mathsf{a.x}$, the direct import induced by a is not involved in the resolution of y that can be resolved normally. Given an incomplete scope graph G, a resolution is valid in this scope if it is valid whatever scopes the scopes variables appearing in the direct imports resolves to.

Formally, given an incomplete scope graph G and a mapping σ from the scope variables of G to concrete scopes, we denote $G.\sigma$ the ground scope graph corresponding to the application of ι to G (substituting scope variable by concrete scopes). We say that a resolution is valid in the incomplete scope graph if it is valid in all its ground instances:

$$\vdash_G x^{\mathsf{R}} \longmapsto x^{\mathsf{D}} \stackrel{def}{=} \forall\, \sigma,\, \vdash_{G.\sigma} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}$$

We probably need to state that "all" the resolution that are valid in all the ground instances are preserved ? We might not notice duplicate definition in an incomplete graph due to this definition.

more likely we want something with sets, let $D_G(x^{\mathsf{R}})$ be the set of possible resolution for x^{R} in G, then given an incomplete graph G:

$$D_G(x^{\mathsf{R}}) \stackrel{def}{=} \begin{cases} D_{G.\bot}(x^{\mathsf{R}}) & \text{if } \forall \sigma\ \sigma', D_{G.\sigma}(x^{\mathsf{R}}) = D_{G.\sigma'}(x^{\mathsf{R}}) \\ \emptyset & \text{else} \end{cases}$$

Or say that x^{R} uniquely resolve in an incomplete graph iff it resolves and it uniquely resolves in all the ground instances. ???

Resolution algorithm in an incomplete graph First take the resolution algorithm and add the direct imports, should be easy to prove correspondence with the calculus with the direct import rule.

Then change the algorithm to handle the variables in direct imports, then prove that on ground instances the algorithm are equal (trivial), then prove that the one with exception respect previous predicate (res in incomplete graph) using the equivalence on ground graphs.

Variables in assumptions The assumptions used to build the scope graphs and the sub-typing relations are not all ground. Some of them can contain variables that also appear in the corresponding set of constraint. Therefore the scope graph and the sub-typing relation are completely defined only when these variable are instantiated. However this does not prevent us to work with the incomplete scope graphs and sub-typing relation:

- For the scope graph, the place a variable can occur in the set of assumption is the left side of a direct import declaration, i.e. an assumption $s_1 \in \mathcal{IS}(s_2)$, s_1 can be a variable. However we have seen in Section 3 that we can still resolve some references in a graph containing incomplete direct imports. And a resolution in such an incomplete graph will still be valid in the final graph that will be an instance of the incomplete one.
- For the sub-typing relation we can also work with a set of assumption containing variables. The corresponding relation is then the transitive closure of the relation whose base cases are only the ground assumptions. This relation will always be a subset of the one corresponding to a ground instance of the set of assumptions

PN ▶*Questions of the day: what happen in the following program:*◀

```
Record A { x : Int}
Record B extends A { x : Bool }

Def y = new B ();
Def z = a.x  + 1
```

PN ▶*And this one:*◀

```
Record A { x : Int}
Record B extends A { x : Bool }

Def y : A = new B ();
Def z = a.x  + 1
```

Don't we want to ensure a property such that:
if $T_1 < T_2 \wedge T_1 \rightsquigarrow S_1 \wedge T_2 \rightsquigarrow S_2$ then
$\vdash_{G[\sigma \mapsto S_1]} r \longmapsto d \implies \vdash_{G[\sigma \mapsto S_2]} r \longmapsto d$??

6.2 Name resolution algorithm

Fig. 6.2 shows the modified name resolution algorithm that works in partial scope graphs.

6.3 Constraint solving algorithm

See Fig. 6.3.

7. Verification

is the resolution sound and complete with respect to the specification; formalize this notion as an extension of the proof of the ESOP paper?

$$\mathit{Res}[\mathbb{I}](x^{\mathsf{R}}) := \{x_i^{\mathsf{D}} \mid \exists S \text{ s.t. } x^{\mathsf{R}} \in \mathcal{R}(S) \wedge x_i^{\mathsf{D}} \in \mathit{Res}_V[\{x^{\mathsf{R}}\} \cup \mathbb{I}, \{\}](x^{\mathsf{R}}, S)$$

$$\mathit{Res}_V[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) := \begin{cases} \mathit{Res}_L[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) \text{ if non-empty} \\ \mathit{Res}_P[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) \end{cases}$$

$$\mathit{Res}_L[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) := \begin{cases} \mathit{Res}_D[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) \text{ if non-empty} \\ \mathit{Res}_I[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) \end{cases}$$

$$\mathit{Res}_D[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \{x_i^{\mathsf{D}} | x_i^{\mathsf{D}} \in \mathcal{D}(S)\} \end{cases}$$

$$\mathit{Res}_I[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \textbf{\textit{exception}} \text{ if } \mathcal{IS}(S) \text{ constains a variable} \\ \bigcup \{x_i^{\mathsf{D}} \mid S' \in IS \wedge x_i^{\mathsf{D}} \in \mathit{Res}_L[\mathbb{I}, \{S\} \cup \mathbb{S}](x^{\mathsf{R}}, S')\} \\ \qquad \text{where } IS := \{S_y \mid y^{\mathsf{R}} \in \mathcal{I}(S) \backslash \mathbb{I} \wedge y^{\mathsf{D}} : S_y \in \mathit{Res}[\mathbb{I}](y^{\mathsf{R}})\} \cup \mathcal{IS}(S) \end{cases}$$

$$\mathit{Res}_P[\mathbb{I}, \mathbb{S}](x^{\mathsf{R}}, S) := \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \mathit{Res}_V[\mathbb{I}, \{S\} \cup \mathbb{S}](x^{\mathsf{R}}, \mathcal{P}(S)) \end{cases}$$

Figure 9. Name resolution algorithm

$x^{\mathsf{R}} \mapsto \delta \wedge C \longrightarrow$	σC *where* $x^{\mathsf{D}} \in$ *Res*$[\![\mathbb{I}]\!](x^{\mathsf{R}})$ *without exception* *with* $\sigma := [\delta \mapsto x^{\mathsf{D}}] \circ \sigma$	(S-RESOLVE)
$t_1 \leq t_2 \wedge C \longrightarrow$	C *where* t_1, t_2 *are not arrows* ***error*** *if* $t_1 \not\leq t_2$	(S-GNDSUBTYPE)
$\texttt{Arrow}(T_1,T_2) \leq \texttt{Arrow}(T_3,T_4) \wedge C \longrightarrow$	$C \wedge T_3 \leq T_1 \wedge T_2 \leq T_4$	(S-ARROWSUBTYPE)
$t \leq \texttt{Arrow}(T_1,T_2) \wedge C \longrightarrow$	***error*** *where* t *is not an arrow*	(S-NOSUBTYPE1)
$\texttt{Arrow}(T_1,T_2) \leq t \wedge C \longrightarrow$	***error*** *where* t *is not an arrow*	(S-NOSUBTYPE2)
$\tau \leq t_1 \wedge \tau \leq t_2 \wedge C \longrightarrow$	$C \wedge \tau \leq (t_1 \sqcap t_2)$ *where* t_1, t_2 *are not arrows* ***error*** *if no greatest lower bound*	(S-GNDUPPERBND)
$\tau \leq \texttt{Arrow}(T_1,T_2) \wedge C \longrightarrow$	$\sigma(\tau \leq \texttt{Arrow}(T_1,T_2) \wedge C)$ *with* $\sigma := [\tau \mapsto \texttt{Arrow}(\tau_1,\tau_2)] \circ \sigma$	(S-ARROWUPPERBND) *new* τ_1, τ_2
$t_1 \leq \tau \wedge t_2 \leq \tau \wedge C \longrightarrow$	$C \wedge (t_1 \sqcup t_2) \leq \tau$ *where* t_1, t_2 *are not arrows* ***error*** *if no least upper bound*	(S-GNDLOWERBND)
$\texttt{Arrow}(T_1,T_2) \leq \tau \wedge C \longrightarrow$	$\sigma(\texttt{Arrow}(T_1,T_2) \leq \tau \wedge C)$ *with* $\sigma := [\tau \mapsto \texttt{Arrow}(\tau_1,\tau_2)] \circ \sigma$	(S-ARROWLOWERBND) *new* τ_1, τ_2
$\tau \leq T \wedge C \longrightarrow$	σC *where* T *is not a variable,* $T' \leq \tau \notin C, \tau \leq T' \notin C$ *with* $\sigma := [\tau \mapsto T] \circ \sigma$	(S-PICKUPPERBND)
$T \leq \tau \wedge C \longrightarrow$	σC *where* T *is not a variable,* $T' \leq \tau \notin C, \tau \leq T' \notin C$ *with* $\sigma := [\tau \mapsto T] \circ \sigma$	(S-PICKLOWERBND)
$T_1 \leq \tau \wedge \tau \leq T_2 \wedge C \longrightarrow$	$\sigma(C \wedge T_1 \leq T_2)$ *where* T_1, T_2 *are not variables,* $T' \leq \tau \notin C, \tau \leq T' \notin C$ *with* $\sigma := [\tau \mapsto T_2] \circ \sigma$	(S-PICKBOTHBND)
$C \longrightarrow$	σC *if no other constraint can be solved and* $[\tau \mapsto T] \notin \sigma$ *with* $\sigma := [\tau \mapsto \top] \circ \sigma$	(S-RESTTOP)

Figure 10. Constraint solving algorithm

8. Validation

LMR and argue why that is representative

9. Related Work (Everyone)

10. Conclusion

Acknowledgments

This research was partially funded by the NWO VICI *Language Designer's Workbench* project (639.023.206). Andrew Tolmach was partly supported by a Digiteo Chair at Laboratoire de Recherche en Informatique, Université Paris-Sud.

References

[1] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[2] P. Neron, A. P. Tolmach, E. Visser, and G. Wachsmuth. A theory of name resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, Lecture Notes in Computer Science. Springer, April 2015. To appear.

[3] B. C. Pierce, editor. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2005.