Submitted to OOPSLA 2015

Type-Dependent Name Resolution

anonymous

Abstract

We extend and combine two existing declarative formalisms, the scope graphs of Neron et al. and type constraint systems, to build a theory that can describe both name and type resolution for realistic languages with complex scope and typing rules. Unlike conventional static semantics presentations, our approach maintains a clear separation between scoping and typing concerns, while still being able to handle language constructs, such as class field access, for which name and type resolution are necessarily intertwined. We define a constraint scheme that can express both typing and name binding constraints, and give a formal notion of constraint satisfiability together with a sound algorithm for finding solutions in important special cases. We describe the details of constraint generation for a model language that illustrates many of the interesting resolution issues associated with modules, classes, and records. Our constraint generator and solver have been implemented and will be submitted as artifacts to accompany the paper.

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. 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.* [9], 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 basic building blocks are *scopes*, which are minimal program 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. 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 resolve to one, none, or many declarations. There is a sound and complete *resolution algorithm* that computes the set of declarations to which each reference resolves.

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]

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 technique goes back at least to Milner's seminal paper on polymorphism [7], and has since been extended to cover many additional language features, notably subtyping. Pottier and Remy (Chapter 10 of [10]) TODO ►*should be a proper citation of the chapter*◄ give a detailed exposition, and show how an efficient resolution

[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.

```
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 1. Program with records

algorithm can be expressed using rewrite rules. The constraint approach is most commonly used for type inference, but even for the simpler problem of type checking, passing to constraints is a useful way to separate the language-dependent part of the task (generating the constraints) from the language-independent part (solving the constaints).

This simple two stage approach—name resolution using the scope graph followed by a separate type resolution stage—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 1, written in a language with nominal records and using standard dot notation for record field access. (Subscripts on identifiers represent source code positions and are not part of the language itself.) In order to resolve the type of $y_7.x_8$ we must first resolve the field name x_8 to the appropriate declaration field (x_2 or x_6). But this name resolution depends on the *type* of y_7, so we must resolve that type first, which again, requires first resolving the *name* of y_7. 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 y_9, then its type, then the name and type of a_{10}, and finally the name and type of x_{11}.

To solve this challenge, we reformulate the task of generating a scope graph from a given program as one of finding 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 1 and for a broad range of other language constructs.

Contributions Our specific contributions are as follows:

- We show how to complement name resolution based on scope graphs with type resolution based on type constraints including type-dependent name resolution (Section 2).
- We extend the name resolution calculus and algorithm of [9] to handle incomplete scope graphs (Section 3, Section 6).
- 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 (Section 4).
- We describe the details of constraint generation for a model language that illustrates many of the interesting resolution issues associated with modules, classes, and records (Section 5).
- We describe an algorithm for solving problems in our constraint language instantiated to use nominal subtyping, and show that it is sound with respect to the satisfiability specification (Section 6).
- (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.

2. Combining Scope Graphs with Types

In this section we describe our approach to type-dependent name resolution using examples in a small model language. We show how scope graphs are used to model name binding, combine scope graphs with type constraints to model type resolution, and discuss extension of the two models to handle type-dependent name resolution.

2.1 Example Language

We illustrate the ideas using LMR (Language with Modules and Records), which extends the LM (Language with Modules) of [9]. The language does not aspire to be a real programming language, but is designed to represent typical and challenging name and type resolution idioms. The grammar of LMR is defined in Fig. 2. The basic features that LMR inherits from LM are:

- Modules and imports: modules can be nested and can import other modules.
- Various flavours of variable binding constructs: variable definitions (**`def`**), first-class functions (**`fun`**), and three flavours of let bindings.
- Declarations (modules, definitions, records) in the same module (scope) are mutually recursive.
- Qualified names allow access to the declaration in a module without import.

LMR extends LM with the following features:

$$
\begin{array}{rcl}
\mathit{prog} & = & \mathit{decl}^* \\
\mathit{decl} & = & \texttt{module}\ \mathit{Id}\ \{\,\mathit{decl}^*\,\} \mid \texttt{import}\ \mathit{Qid} \mid \texttt{def}\ \mathit{bind} \\
 & \mid & \texttt{record}\ \mathit{Id}\ \mathit{sup}?\ \{\mathit{fdecl}^*\} \\
\mathit{sup} & = & \texttt{extends}\ \mathit{Qid} \\
\mathit{fdecl} & = & \mathit{id}\ \texttt{:}\ \mathit{ty} \\
\mathit{ty} & = & \texttt{Int} \mid \mathit{Qid} \mid \mathit{ty} \to \mathit{ty} \\
\mathit{exp} & = & \mathit{int} \mid \texttt{true} \mid \texttt{false} \mid \mathit{qid} \mid \mathit{exp} \oplus \mathit{exp} \\
 & \mid & \texttt{if}\ \mathit{exp}\ \texttt{then}\ \mathit{exp}\ \texttt{else}\ \mathit{exp} \\
 & \mid & \texttt{fun}\ (\ \mathit{id}\ \texttt{:}\ \mathit{ty}\ )\ \{\,\mathit{exp}\,\} \mid \mathit{exp}\ \mathit{exp} \\
 & \mid & \texttt{let}\ \mathit{bind}^*\ \texttt{in}\ \mathit{exp} \mid \texttt{letpar}\ \mathit{bind}^*\ \texttt{in}\ \mathit{exp} \\
 & \mid & \texttt{letrec}\ \mathit{tbind}^*\ \texttt{in}\ \mathit{exp} \\
 & \mid & \texttt{new}\ \mathit{Qid}\ \{\mathit{fbind}^*\} \mid \texttt{with}\ \mathit{exp}\ \texttt{do}\ \mathit{exp} \\
 & \mid & \mathit{exp}\ \texttt{.}\ \mathit{id} \\
\mathit{Qid} & = & \mathit{Id} \mid \mathit{Qid}\ \texttt{.}\ \mathit{Id} \\
\mathit{qid} & = & \mathit{id} \mid \mathit{Qid}\ \texttt{.}\ \mathit{id} \\
\mathit{bind} & = & \mathit{id} = \mathit{exp} \mid \mathit{tbind} \\
\mathit{tbind} & = & \mathit{id}\ \texttt{:}\ \mathit{ty} = \mathit{exp} \\
\mathit{fbind} & = & \mathit{id} = \mathit{exp}
\end{array}
$$

Figure 2. Syntax of LMR.

- LMR is statically typed: function arguments require explicit type annotations, but bindings of variables may be left for type inference to resolve.
- Declaration of nominal record types with inheritance and a corresponding subtyping relation on record types.
- Construction of records with **`new`** using references to fields for initialization.
- Access to the fields of a record value using dot notation `e.f`.
- Implicit access to record fields using a Pascal-like **`with`** construct.

In the rest of this section we study name and type resolution for a selection of LMR constructs that explain the ideas of type-dependent name resolution using examples. In Section 5 we give a complete account of extraction of constraints for all LMR constructs. In Section 3 we present the formal rules of the resolution calculus that define name resolution in a scope graph.

2.2 Declarations and References

We recall the concepts of the scope graph approach [9] and extend it with type constraints. Consider the example in Fig. 2.2, which shows an LMR program (top), and the scope graph diagram (right) and the constraints (left) extracted from it. Subscripts on identifiers represent AST positions. Thus, $\mathtt{x}_1$ and $\mathtt{x}_3$ are different occurrences of the *same* name `x`.

Scope Graph TODO ▶*You should note somewhere in the example explanation that the def's are mutually recursive.*◀

The key building block of a scope graph is the *scope*, an abstraction of a set of nodes in the AST that behave uniformly with respect to name binding. In a scope graph diagram, scopes are represented by circles with numbers

```
def x1 = 1
def y2 = if x3 == 0 then 3 else x4
```

declarations

$\mathtt{x}_1^{\mathsf{D}} : \tau_1$
$\mathtt{y}_2^{\mathsf{D}} : \tau_2$

references

$\mathtt{x}_3^{\mathsf{R}} \mapsto \delta_1$
$\delta_1 : \mathit{Int}$
$\mathtt{x}_4^{\mathsf{R}} \mapsto \delta_2$
$\delta_2 : \tau_4$

types

$\tau_1 \equiv \mathit{Int}$
$\mathit{Bool} \equiv \mathit{Bool}$
$\mathit{Int} \equiv \mathit{Int}$
$\tau_3 \equiv \mathit{Int}$
$\tau_2\ \mathit{is}\ \tau_3 \sqcup \tau_4$

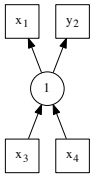

Figure 3. Definition and reference

representing their identity. Scopes manage the visibility of *declarations*. In a diagram, declarations are represented by boxes with an *incoming arrow* from a scope. In the example program $\mathtt{x}_1$ and $\mathtt{y}_2$ are declarations. In constraints we denote declarations using a D superscript (e.g. $\mathtt{x}_1^{\mathsf{D}}$). *References* are identifiers that refer to a declaration. In diagrams, a reference is represented by means of a box with an arrow pointing to its scope. In the program $\mathtt{x}_3$ and $\mathtt{x}_4$ are references. In constraints we denote references with an R superscript (e.g. $\mathtt{x}_3^{\mathsf{R}}$). *Name resolution* in a scope graph consists of finding a path in the scope graph from a reference to a declaration. Since scope 1 contains a declaration $\mathtt{x}_1^{\mathsf{D}}$ with the name `x`, both references $\mathtt{x}_3^{\mathsf{R}}$ and $\mathtt{x}_4^{\mathsf{R}}$ resolve to the declaration $\mathtt{x}_1^{\mathsf{D}}$, which we write $\mathtt{x}_3^{\mathsf{R}} \longmapsto \mathtt{x}_1^{\mathsf{D}}$.

Type Constraints Scope graphs do not include explicit type information. However, by associating type information 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. However, that requires a language dependent mechanism. In order to abstract from the language-specific representation of type information in the AST, we generate constraints in a language-independent constraint language, as illustrated in the left box of Fig. 2.2.

The constraints can be categorized into three groups. *Declaration constraints* associate types with declarations. In the example, the constraints $\mathtt{x}_1^{\mathsf{D}} : \tau_1$ and $\mathtt{y}_2^{\mathsf{D}} : \tau_2$ associate type variables with declarations $\mathtt{x}_1^{\mathsf{D}}$ and $\mathtt{y}_2^{\mathsf{D}}$. *Reference constraints* retrieve the types of variables by means of a resolution constraint associating a declaration variable to a reference, and a type association constraint for the declaration variable. For example, the constraint $\mathtt{x}_3^{\mathsf{R}} \mapsto \delta_1$ resolves

reference x_3^R to declaration variable δ_1, and the constraint $\delta_1 : Int$ requires the type of that declaration to be Int because of the use of the reference in the equality operator. Finally, *type constraints* pose equality and subtype constraints on the types assigned to declarations and expressions. For example, the constraint $\tau_1 \equiv Int$ arises from the declaration of x_1^D, the constraint $Bool \equiv Bool$ arises from the condition of the **if**, the constraint $Int \equiv Int$ arises from the `0` argument of the equality, and $\tau_3 \equiv Int$ arises from the integer `3`. (We will leave the trivial equality constraints out in further examples.) Finally, the branches of the **if** generate a least upper-bound constraint τ_2 *is* $\tau_3 \sqcup \tau_4$ on the types of the branches.

Resolution The combination of a scope graph and type constraints define a *resolution problem*. A solution for such a problem is a substitution for the declaration and type variables in the problem such that (1) name resolutions are consistent with the scope graph according to the rules of the resolution calculus (Section 3), and (2) all type constraints are satisfied. For the example, the solution TODO ▶*add generated solution to the figure*◀.

2.3 Lexical Scope and Subtypes

Fig. 2.3 shows a larger LMR example that illustrates lexical scope and subtype constraints.

Lexical scope is modeled using parent edges between scopes in the scope graph. In the example, scope 3, corresponding to the body of the **fun**, is enclosed in scope 2, corresponding to the **letrec**, which is enclosed in scope 1, the global scope of the program. Resolution of a reference proceeds from the scope of the reference to parent scopes until a matching declaration is found. Thus, reference n_5^R resolves to declaration n_3^D, which shadows n_1^D.

A function application such as $fac_6(n_7 - 1)$ requires that the type of the actual parameter (τ_7) is a subtype of the type of the formal parameter (τ_6).

2.4 Imports

In addition to lexical scope, many programming languages provide features for making declarations in scopes selectively available 'at a distance'. Examples of such constructs are modules with imports in ML and classes with inheritance in Java. To model such features, scope graphs provide *associated scopes* and *imports*.

Associated Scope The LMR program in Fig. 5 consists of two *modules* A_1 and B_3 and an import from the former in the latter. The declarations in these modules are contained in scopes 2 and 3, which are child scopes of the root scope 1. These scopes are *associated* with the declaration of the name of the module, which is represented in a scope graph diagram with an open arrow from the declaration (e.g. A_1^D) to the scope (e.g. 2).

Imports The declarations in a scope are only visible to references in lexically enclosed scopes, i.e. following parent

```
def n1 =
  letrec fac2 : Int →Int = fun (n3: Int) {
           if (n4 == 0) then 1
           else n5 * fac6(n7 − 1) }
    in fac8(5)
```

declarations

$n_1^D : \tau_1$
$fac_2^D : Fun[Int,Int]$
$n_3^D : Int$

references

$n_4^R \mapsto \delta_1$
$\delta_1 : Int$
$n_5^R \mapsto \delta_2$
$\delta_2 : Int$
$fac_6^R \mapsto \delta_3$
$\delta_3 : Fun[\tau_6,Int]$
$n_7^R \mapsto \delta_4$
$\delta_4 : Int$
$fac_8^R \mapsto \delta_5$
$\delta_5 : Fun[\tau_8,\tau_1]$

types

$\tau_2 \preceq Fun[Int,Int]$
$\tau_2 \equiv Fun[Int,\tau_3]$
$Bool \equiv Bool$
$Int \equiv Int$
$\tau_4 \equiv Int$
$\tau_5 \equiv Int$
$\tau_7 \equiv Int$
$Int \equiv Int$
$\tau_7 \preceq \tau_6$
τ_3 *is* $\tau_4 \sqcup \tau_5$
$\tau_9 \equiv Int$
$\tau_9 \preceq \tau_8$

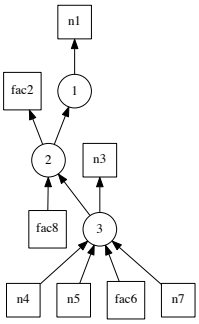

Figure 4. Scope

edges to child scopes. An *import* makes the declarations in a scope visible in another, not necessarily lexically related, scope. An import is represented by (1) a regular reference of the name in its enclosing scope, and (2) an import in that scope. The latter is represented using an open arrow from a scope to a reference. For example, **import** A_4 is represented by the reference A_4^R in scope 3 and an import arrow from scope 3 to A_4^R.

Resolving through Imports Name resolution in the presence of associated scopes and imports proceeds as follows. If a scope S_1 contains an import x_i^R, which resolves to a declaration x_j^D with associated scope S_2, then all declarations in S_2 are reachable in S_1. Thus, in the example, reference a_6^R resolves to declaration a_2^D since the import A_4^R resolves to declaration A_1^D, and the associated scope 2 of A_1^D contains declaration a_2^D. Note that the resolution calculus is param-

```
module A1 {
  def a2 = 4
}
module B3 {
  import A4
  def b5 = a6 + 56
}
```

declarations

$a_2^{\mathsf{D}} : \tau_1$
$b_5^{\mathsf{D}} : \tau_2$

references

$a_6^{\mathsf{R}} \mapsto \delta_1$
$\delta_1 \ : \ Int$

types

$\tau_1 \equiv Int$
$\tau_2 \equiv Int$
$Int \equiv Int$

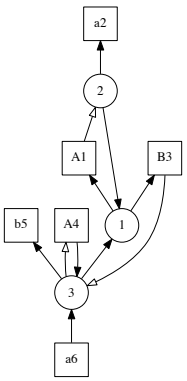

Figure 5. Imports

```
module A1 {
  def x2 = 79
}
def y3 = A4.x5
```

declarations

$x_2^{\mathsf{D}} : \tau_1$
$y_3^{\mathsf{D}} : \tau_2$

references

$x_5^{\mathsf{R}} \mapsto \delta_1$
$\delta_1 \ : \ \tau_2$

types

$\tau_1 \equiv Int$

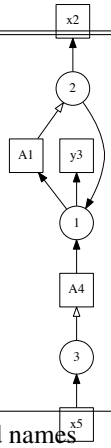

Figure 6. Qualified names

eterized with the policy to disambiguate conflicting resolutions. Here we use the default policy of [9] that prefers imported declarations over declarations in parents.

Qualified Names Another common pattern for accessing the declarations in a scope is through qualified names. Instead of importing *all* declarations in a scope, a single declaration is accessed. For example, in Fig. 6 the expression $\texttt{A}_4\texttt{.x}_5$ refers to the declaration $\texttt{x}_2^{\mathsf{D}}$ in module $\texttt{A}_1$. This pattern can be modeled using the scope graph ingredients that we have seen so far. The reference $\texttt{x}_5^{\mathsf{R}}$ is defined as a reference of scope 3, which has no parent. The only declarations visible in scope 3 are through the import of $\texttt{A}_4^{\mathsf{R}}$, which is itself a reference in scope 1. Thus, resolution of $\texttt{A}_4^{\mathsf{R}}$ leads to

```
record A1 { x2: Int }
def a3 = new A4 { x5 = 1 }
def y6 = with a7 do x8 + 1
```

declarations

$x_2^{\mathsf{D}} : Int$
$a_3^{\mathsf{D}} : \tau_1$
$y_6^{\mathsf{D}} : \tau_4$

references

$A_4^{\mathsf{R}} \mapsto \delta_1$
$x_5^{\mathsf{R}} \mapsto \delta_2$
$\delta_2 \ : \ \tau_2$
$a_7^{\mathsf{R}} \mapsto \delta_4$
$\delta_4 \ : \ Rec(\delta_3)$
$\delta_3 \rightsquigarrow \varsigma_1$
$x_8^{\mathsf{R}} \mapsto \delta_5$
$\delta_5 \ : \ Int$

types

$Rec(A_1^{\mathsf{D}}) <: Rec(A_1^{\mathsf{D}})$
$\tau_1 \equiv Rec(\delta_1)$
$\tau_3 \preceq \tau_2$
$\tau_3 \equiv Int$
$\tau_4 \equiv Int$
$Int \equiv Int$

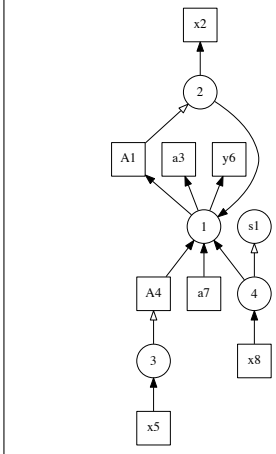

Figure 7. With

$\texttt{A}_1^{\mathsf{D}}$ and therefore the declarations in associated scope 2 are visible in scope 3.

2.5 Type-Dependent Name Resolution

However, for some language constructs the resolution of a name to its declaration depends on the type of another expression.

For example, in 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.

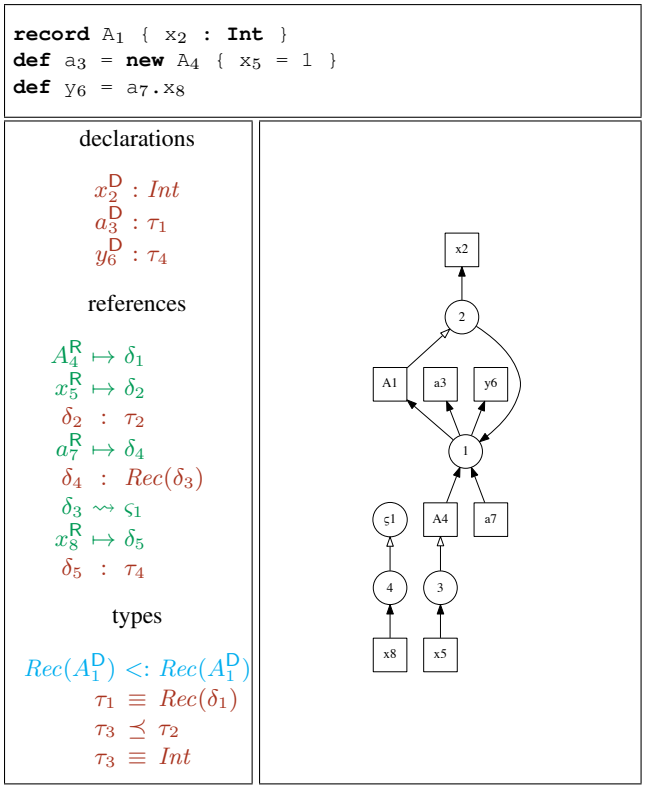

Figure 8. Field access

3. Extended scope graphs

TODO ►*explain why this model is interesting and emphasizethat it covers difficult problems such as imports and inheritance*◄

3.1 Scope Graphs

A *scope graph*, introduced by Neron et al. [9], is a generic model for representing the name binding structure of programs. A scope graph $\mathcal{G}$ is built around three basic types of nodes derived from the program abstract syntax tree (AST), *declarations, references, and scopes*:

- A *declaration* is an occurrence of an identifier that introduces a name. x_i^{D} denotes the definition of name x at position i in the program. We omit the position i when this can be inferred from context. $\mathcal{D}(\mathcal{G})$ denotes the set of references of $\mathcal{G}$.
- A *reference* is an occurrence of an identifier referring to a declaration. We write x_i^{R} for a reference with name x at position i. Again, the position i may be omitted when it can be inferred from context. $\mathcal{R}(\mathcal{G})$ denotes the set of references of $\mathcal{G}$.
- A *scope* is an abstraction over a set of nodes in the AST that behave uniformly with respect to name binding. $\mathcal{S}(\mathcal{G})$ denotes the set of scopes of $\mathcal{G}$.

Given these sets, a scope graph is defined by the following functions:

- Each declaration d in $\mathcal{D}(\mathcal{G})$ is declared within a scope denoted $\mathcal{S}c(d)$.
- Each declaration d has an optional *associated scope*, $\mathcal{DS}c(d)$ that is the scope corresponding to the body of the declaration. For example, the declarations in a module are elements of its associated scope.
- Each reference r in $\mathcal{R}(\mathcal{G})$ is declared within a scope denoted $\mathcal{S}c(r)$.
- Each scope S in $\mathcal{S}(\mathcal{G})$ has an optional *parent scope* $\mathcal{P}(S)$ that corresponds to its *lexically enclosing scope*. The parent relation has to be well-founded, i.e. there is no infinite sequence S_n such that $S_{n+1} = \mathcal{P}(S_n)$.
- Each scope S has an associated set of references $\mathcal{I}(S)$, that represents the *imports* in this scope

We define by comprehension the set of declarations enclosed in a scope S, as $\mathcal{D}(S) = \{d \mid \mathcal{S}c(d) = S\}$.

3.2 Resolution Calculus

Given this model, the *resolution calculus* defines the *resolution* of a reference to a declaration in a scope graph [9] as the minimal path from reference to declaration through parent and import edges. A path p is a list of steps representing the atomic 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 decla-

ration step $\mathbf{D}(x^{\mathsf{D}})$ leading to a declaration x^{D}. Given a seen import 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:

$$\mathbb{I} \vdash p : x_i^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$$

The calculus in Fig. 10 defines the resolution relation in terms of edges in the scope graph, reachable declarations, and visible declarations.

The resolution calculus is parametrized by two predicates on paths, a *well-formedness predicate* $WF(p)$ and an *ordering relation* $<$ that allows the formalization of different name-binding policies such as transitive vs non-transitive imports. A typical definition of the well-formedness predicate is *no-parents-after-imports*, which entails that a resolution can not proceed to a lexical parent after an import transition. Fig. 9 presents the definition of paths (p) consisting of steps (s) and an example of a path well-formedness predicate and path ordering relation. This configuration supports arbitrary levels of lexical scope ($\mathbf{P}^*$), transitive imports ($\mathbf{I}(\_)^*$), no-parents-after-imports (an $\mathbf{I}(\_)$ step cannot be followed by a $\mathbf{P}$), prefer local declarations over imported declarations (DI), prefer local declarations over declarations in parents (DP), and prefer imported declarations over declarations in parents (IP).

TODO ►*an example here of basic resolution calculus*◄

TODO ►*also introduce the graphical notation of scope graph diagrams*◄

3.3 Direct Imports

The scope graph model and resolution calculus provide a large coverage of binding constructors in existing programming languages. However, it does not support the dependence of resolution of a reference on the *type* of an expression. For example, in an object-oriented language, the name `x` in `e.x` has to be resolved to the declaration of the field `x` that appears in some class definition `A` that is the type of the receiver expression `e`.

TODO ► *Another example is Pascal's* with *constructor (as introduced in ...); probably refer to figure with example* ◄

In order to model such constructs we extend the scope graph with *direct imports*. A direct import defines a direct link between two scopes without the use of a reference. In addition to its set of associated imports (references of the form x^{R}), a scope is extended with an associated set of direct imports $\mathcal{IS}(S)$ consisting of other scopes in the graph. For these imports we introduce the (D) scope transition rule, which is similar to the (I) rule of the original calculus, except that this transition does not require the intermediate resolution of a reference:

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

The complete resolution calculus with this new rule is presented in Figure 10.

Resolution paths

$$\begin{array}{rl} s & := \mathbf{D}(x_i^{\mathsf{D}}) \mid \mathbf{I}(x_i^{\mathsf{R}}, x_j^{\mathsf{D}}) \mid \mathbf{I}(S) \mid \mathbf{P} \\ p & := [] \mid s \mid p \cdot p \\ & \text{(inductively generated)} \\ [] \cdot p & = p \cdot [] = p \\ (p_1 \cdot p_2) & \cdot p_3 = p_1 \cdot (p_2 \cdot p_3) \end{array}$$

Well-formed paths

$$WF(p) \Leftrightarrow p \in \mathbf{P}^* \cdot \mathbf{I}(\_)^*$$

Specificity ordering on paths

$$\frac{}{\mathbf{D}(\_) < \mathbf{I}(\_)} \ (DI) \qquad \frac{}{\mathbf{I}(\_) < \mathbf{P}} \ (IP) \qquad \frac{}{\mathbf{D}(\_) < \mathbf{P}} \ (DP)$$

$$\frac{s_1 < s_2}{s_1 \cdot p_1 < s_2 \cdot p_2} \ (Lex1) \qquad \frac{p_1 < p_2}{s \cdot p_1 < s \cdot p_2} \ (Lex2)$$

Figure 9. Resolution paths, well-formedness predicate, and specificity ordering as introduced in [9]

Edges in scope graph

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

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

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

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

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

Reference resolution

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

Figure 10. Resolution calculus from [9] extended with direct import rule D

4. Constraint Language

In this section we introduce the syntax and declarative semantics of constraints.

4.1 Syntax of Contraints

Fig. 11 defines the syntax of constraints. The language independent base terms of the constraint language are:

- *Declarations* in D, which are either ground declarations x_i^{D} of the program or variables δ
- *References* in R, which are either ground references x_i^{R} of the program or variables ρ
- *Scopes* in S, which are either ground scopes of the program denoted S_i or variables ς
- *Positions* in I, which can be ground positions i, positions of a declaration or a reference $Pos(d)/Pos(r)$ or variables ι
- *Types* in T, which are either type variables τ or type constructor applications $c(\mathrm{T}, ..., \mathrm{T})$ with $c \in C_{\mathcal{T}}$, the set of language-specific type constructors.

Given these terms we define the syntax of constraints, which come in two flavors, *assumptions* and proper *constraints*. *Assumptions*, defined by the sort A, correspond to known facts about a program: the scope of a reference ($\mathcal{S}c(\mathrm{R}) := \mathrm{S}$), the scope of a declaration ($\mathcal{S}c(\mathrm{D}) := \mathrm{S}$), the associated scope of a declaration ($\mathrm{D} \rightsquigarrow \mathrm{S}$), the parent of a scope ($P(\mathrm{S}) := \mathrm{S}$), a named import in a scope ($\mathrm{S} \in \mathcal{I}(\mathrm{S})$), a direct import in a scope ($\mathrm{S} \in \mathcal{IS}(\mathrm{S})$), and a subtype relation between types ($\mathrm{T} <: \mathrm{T}$). *Constraints*, defined by the sort C, represent the restrictions on name and type resolution, which consist of: resolution of a reference to a declaration ($\mathrm{R} \mapsto \mathrm{D}$), equality of two types ($\mathrm{T} \equiv \mathrm{T}$), subtype relation between two types ($\mathrm{T} \preceq \mathrm{T}$), associated scope of a declaration ($\mathrm{D} \rightsquigarrow \mathrm{S}$), the type of a declaration ($\mathrm{D} : \mathrm{T}$), and the least upper bound of two sorts ($\mathrm{T}\ \mathit{is}\ \mathrm{T} \sqcup \mathrm{T}$). Assumptions and constraints can be combined using conjunction ($\mathrm{C} \wedge \mathrm{C}$) and True represents the absence of constraints.

Language-Specific Types The language of constraints defined above is independent of the language under analysis, except for the type constructors introduced by the language.

The type constructors in $C_{\mathcal{T}}$

type constructors have an arity $c : n$

example type constructors such as Int, Bool, and Arrow;

type constructors can also have more structure

to represent user-defined types, a type constructor includes the identity of the type definition

we use declarations (in the formal sense of scope graphs) to represent this identity

for example, record types in LMR are represented by $Rec(d)$ with d a declaration in the program

Thus, in program Fig. **??**, the record definition A_1 defines the type $Rec(A_1^{\mathsf{D}})$

$$
\begin{array}{llll}
\mathrm{A} := & \mathcal{S}c(\mathrm{R}) := \mathrm{S} & \mid \mathrm{D} \rightsquigarrow \mathrm{S} & \mid \mathrm{S} \in \mathcal{I}(\mathrm{S}) \\
\mid & \mathcal{S}c(\mathrm{D}) := \mathrm{S} & \mid P(\mathrm{S}) := \mathrm{S} & \mid \mathrm{S} \in \mathcal{IS}(\mathrm{S}) \\
\mid & \mathrm{T} <: \mathrm{T} & & \\
\\
\mathrm{C} := & \mathsf{True} & \mid \mathrm{R} \mapsto \mathrm{D} & \mid \mathrm{T} \equiv \mathrm{T} \\
\mid & \mathrm{C} \wedge \mathrm{C} & \mid \mathrm{D} \rightsquigarrow \mathrm{S} & \mid \mathrm{T} \preceq \mathrm{T} \\
\mid & \mathrm{D} : \mathrm{T} & \mid \mathrm{T}\ \mathit{is}\ \mathrm{T} \sqcup \mathrm{T} & \mid \mathrm{A} \\
\mathrm{D} := & \delta \mid x_i^{\mathsf{D}} & \mathrm{S} := \varsigma \mid S_i \mid \bot & \\
\mathrm{R} := & \rho \mid x_i^{\mathsf{R}} & \mathrm{T} := \tau \mid c(\mathrm{T}, ..., \mathrm{T}) \text{ with } c \in C_{\mathcal{T}} &
\end{array}
$$

Figure 11. Syntax of constraints

languages with user-defined types, such as classes in OO language or algebraic data types in functional languages,

identity of user-defined types

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**◀. *o*
- ***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 Constraints

In our approach, the abstract syntax tree of a program p is reduced by a language-specific extraction function $[\![p]\!]$ to a constraint following the syntax defined in Fig. 11. Given commutativity and associativity of conjuction, such a constraint is equivalent to a constraint of the form

$$A_1 \wedge \ldots \wedge A_n \wedge C_1 \wedge \ldots \wedge C_m$$

consisting of a set of assumptions A_i and a set of constraints C_j. (We define an example extraction function in the next section.) The assumptions define the scope graph and subtyping relation with respect to which the other constraints need to be solved.

4.2.1 Interpretation of Assumptions

We denote with $A^{<:}$ the set of assumptions of the form $T_1 <: T_2$ in A that will be used to build the corresponding sub-typing relation. We denote with $A^{\mathcal{G}}$ the subset formed by the other assumptions that will define the scope graph of the

program. Given a ground set of assumption A, we denote $|A|$ the interpretation of A as the pair $\mathcal{G}, \leq$ where $\leq$ is the sub-typing relation derived from $A^{<:}$ and $\mathcal{G}$ is the scope graph derived from $A^{\mathcal{G}}$.

Sub-typing From the set of assumptions $A^{<:}$ we derive the relation $\leq$ between ground types, built using the type constructors in $C_{\mathcal{T}}$. A variance v is a non-empty subset of $\{-,+\}$ written $-$ for contravariant, $+$ for covariant and $\pm$ for invariant. We also denote $\leq$ for $\leq^{+}$, $\geq$ for $\leq^{-}$ and $=$ for $\leq^{\pm}$. We require all the arguments in the signature of a constructor c to be annotated with a variance parameter. Thus the signature of a type constructor c is declared as $c :: v_1 * ... * v_n$, with the v_i variance annotations. Given such a signature for all the type constructors, we now define the sub-typing relation $\leq$ derived from a set $A^{<:}$ of sub-typing assumptions by the following inductive rules:

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

$$\frac{T_1 <: T_2 \in A^{<:}}{T_1 \leq T_2} \qquad \frac{c :: v_1 * ... * v_n \quad \forall i,\ s_i \leq^{v_i} t_i}{c(s_1, ..., s_n) \leq c(t_1, ..., t_n)}$$

Scope graph The assumptions that are not sub-typing assumptions define the scope graph of the program. These assumptions define the set of scopes, declarations and references and the corresponding relations as follows:

- $P(S) := S'$ defines a new scope S and declares its parent $\mathcal{P}(S)$ as S' when S' is not $\bot$
- $\mathcal{S}c(x^{\mathsf{D}}) := S$ defines a new declaration x^{D} and declares its enclosing scope $\mathcal{S}c(x^{\mathsf{D}})$ as S
- $d \rightsquigarrow S$ declares scope S as the associated scope $\mathcal{DS}c(d)$ of declaration d
- $\mathcal{S}c(x^{\mathsf{R}}) := S$ defines a new reference x^{R} and declares its enclosing scope $\mathcal{S}c(x^{\mathsf{R}})$ as S
- $r \in \mathcal{I}(S)$ adds the reference r to the set of named imports $\mathcal{I}(S)$ of scope S
- $S' \in \mathcal{IS}(S)$ adds the scope S' to the set of direct imports $\mathcal{I}(S)$ of scope S

The result is a correct scope graph according to Section 3 provided that the parent relation is well-founded.

4.2.2 Interpretation of Constraints

The interpretation of a non-assumption constraint (which we will just call constraints) is defined as a truth value in a context, which is a triple of the following elements:

- A scope graph $\mathcal{G}$, as defined in Section 3
- A sub-typing relation $\leq$ on ground types
- A typing environment ψ mapping declarations in $\mathcal{D}(\mathcal{G})$ to types in T

A context $\mathcal{G}, \leq, \psi$ satisfies a constraint C if the predicate $\mathcal{G}, \leq, \psi \models C$ holds. This predicate is defined by the set of

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

$$\frac{\mathcal{G}, \leq, \psi \models C_1 \quad \mathcal{G}, \leq, \psi \models C_2}{\mathcal{G}, \leq, \psi \models C_1 \wedge C_2} \quad \text{(C-And)}$$

$$\frac{\psi(d) = T}{\mathcal{G}, \leq, \psi \models d : T} \quad \text{(C-TypeOf)}$$

$$\frac{\vdash_{\mathcal{G}} p : x^R_{i_1} \mapsto x^D_{i_2}}{\mathcal{G}, \leq, \psi \models r \mapsto d} \quad \text{(C-Resolve)}$$

$$\frac{\mathcal{DS}c(d) = S}{\mathcal{G}, \leq, \psi \models d \rightsquigarrow S} \quad \text{(C-ScopeOf)}$$

$$\frac{t_1 = t_2}{\mathcal{G}, \leq, \psi \models t_1 \equiv t_2} \quad \text{(C-Eq)}$$

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

$$\frac{T = \sqcup_{\leq}\{T_1, T_2\}}{\mathcal{G}, \leq, \psi \models T \textit{ is } T_1 \sqcup T_2} \quad \text{(C-Lub)}$$

Figure 12. Interpretation of pure constraints

inductive rules in Fig. 12, where $=$ is the syntactic equality on terms, and, when it exists, $\sqcup_S$ denotes the least upper bound of types in S according to order $\leq$.

4.3 Program Resolution

The goal of the resolution of the program p is to build a multi-sorted substitution ϕ and a typing environment ψ such that, if $[\![p]\!] = A_p \wedge C_p$ then the following property holds:

$$|\phi(A_p)|, \psi \models \phi(C_p) \qquad (\diamond)$$

TODO ▶*Change notation for interpretation of set of assumptions*◀ Where $\phi(E)$ denotes the application of the substitution ϕ to all the variables appearing in E that are in the domain of ϕ. Note that ϕ has to make $\phi(A_p)$ a set of ground assumptions in order to be able to interpret it whereas some free variable may remain in $\phi(C_p)$. When the proposition $\diamond$ holds we say that ψ and ϕ resolve p.

5. Constraint Collection

In this section, we show how to collect constraints for name resolution and typechecking from programs in the LMR language, whose concrete syntax was given in Figure **??**. The full collection algorithm is shown in Figures 13 and 14. Collection is performed by a single traversal over the program that collects scope and subtyping assumptions, name resolution constraints, and typing constraints all in one pass. (The color codings should help in distinguishing these different kinds of constraints.)

To simplify and compress the presentation, we describe the algorithm as operating over LMR's concrete syntax.[2] The algorithm is defined by a family of functions indexed by syntactic category (*decl*, *exp*, etc.). Each function takes a syntactic item and possibly one or more auxiliary parameters, and (usually) returns a constraint, possibly involving one or more fresh variables or new scope identifiers. Functions are defined by a set of rules, one for each possible syntactic form in the category. For example, $[\![-]\!]^{decl}_{s}$ has four rules (for **module**, **import**, **def** and **record** declarations, respectively), and is parameterized by the scope s into which declared identifiers are to be installed; it returns the conjunction of constraints that enforces correct name and type resolution for the declaration, some of which are derived by invoking generation functions on syntactic sub-components.

To further streamline the presentation, we use the notation $[\![-]\!]^{c^*}$ on sequences of items of syntactic category c to mean the result of applying $[\![-]\!]^{c}$ to each item and returning the conjunction of the resulting constraints, or True for the empty sequence. Similarly, $[\![-]\!]^{c?}$ works on a optional c item; it applies $[\![-]\!]^{c}$ to the item if it is present and returns True otherwise. Throughout, we use metavariable x_i for a (lower case) term variable at position i and X_i for an (upper case) module or record name at position i, with one exception: for compactness, we give just one rule for both qid and Qid, in which x_i can be either kind of identifier. We write Xs for a dot-separated sequence of module or record names, which can be empty (in which case, by convention, $Xs.x$ doesn't have a leading dot).

Let us trace how the constraint generator works on some of the different syntactic forms of LMR. A complete program is a sequence of mutually-recursive top-level declarations, so $[\![-]\!]^{prog}$ creates a new root scope S in which they are to be installed, generates an assumption constraint that S is parentless, and then conjoins the constraints for each declaration, passing S as a parameter to the declaration generator for *decl*. $[\![-]\!]^{decl}_{s}$ generates an assumption that installs any declared identifier into scope s; the rest of its behavior depends on the kind of declaration:

$$
\begin{array}{rcl}
[\![ds]\!]^{prog} & := & P(S) := \bot \quad \textit{(new } S\textit{)} \\
 & \wedge & [\![ds]\!]^{decl^*}_{S} \\
[\![\texttt{module}\ X_i\ \{\,ds\,\}]\!]^{decl}_{s} & := & \mathcal{S}c(X_i^{\mathsf{D}}) := s \quad \textit{(new } S'\textit{)} \\
 & \wedge & P(S') := s \\
 & \wedge & X_i^{\mathsf{D}} \rightsquigarrow S' \ \wedge\ [\![ds]\!]^{decl^*}_{S'} \\
[\![\texttt{import}\ Xs\,.\,X_i]\!]^{decl}_{s} & := & X_i^{\mathsf{R}} \in \mathcal{I}(s) \ \wedge\ [\![Xs\,.\,X_i]\!]^{qid}_{s} \\
[\![\texttt{def}\ b]\!]^{decl}_{s} & := & [\![b]\!]^{bind}_{s,s} \\
[\![x_i = e]\!]^{bind}_{s_r,s_d} & := & \mathcal{S}c(x_i^{\mathsf{D}}) := s_d \quad \textit{(fresh } \tau\textit{)} \\
 & \wedge & x_i^{\mathsf{D}} : \tau \ \wedge\ [\![e]\!]^{exp}_{s_r,\tau} \\
[\![x_i\ \texttt{:}\ t = e]\!]^{bind}_{s_r,s_d} & := & \mathcal{S}c(x_i^{\mathsf{D}}) := s_d \quad \textit{(fresh } \tau\textit{)} \\
 & \wedge & x_i^{\mathsf{D}} : t' \ \wedge\ \tau \preceq t' \\
 & \wedge & C \ \wedge\ [\![e]\!]^{exp}_{s_r,\tau} \\
 & & \textit{where } [\![t]\!]^{ty}_{s_r} = (t', C) \\
[\![\texttt{record}\ X_i\ u\ \{fs\}]\!]^{decl}_{s} & := & \mathcal{S}c(X_i^{\mathsf{D}}) := s \quad \textit{(new } S'\textit{)} \\
 & \wedge & P(S') := s \\
 & \wedge & X_i^{\mathsf{D}} \rightsquigarrow S' \\
 & \wedge & [\![u]\!]^{sup?}_{s,S',Rec(X_i^{\mathsf{D}})} \ \wedge\ [\![fs]\!]^{fdecl^*}_{s,S'} \\
[\![\texttt{extends}\ Xs\,.\,X_i]\!]^{sup}_{s_r,s_d,t} & := & X_i^{\mathsf{R}} \in \mathcal{I}(s_d) \quad \textit{(fresh } \delta\textit{)} \\
 & \wedge & X_i^{\mathsf{R}} \mapsto \delta \\
 & \wedge & t <: Rec(\delta) \ \wedge\ [\![Xs\,.\,X_i]\!]^{qid}_{s_r} \\
[\![x_i\texttt{:}t]\!]^{fdecl}_{s_r,s_d} & := & \mathcal{S}c(x_i^{\mathsf{D}}) := s_d \\
 & \wedge & x_i^{\mathsf{D}} : t' \ \wedge\ C \\
 & & \textit{where } [\![t]\!]^{ty}_{s_r} = (t', C) \\
[\![x_i]\!]^{qid}_{s} & := & \mathcal{S}c(x_i^{\mathsf{R}}) := s \\
[\![Xs\,.\,X_j\,.\,x_i]\!]^{qid}_{s} & := & \mathcal{S}c(x_i^{\mathsf{R}}) := S' \quad \textit{(new } S'\textit{)} \\
 & \wedge & P(S') := \bot \\
 & \wedge & X_j^{\mathsf{R}} \in \mathcal{I}(S') \ \wedge\ [\![Xs\,.\,X_j]\!]^{qid}_{s} \\
[\![\texttt{fun}\ f\ (x_i\texttt{:}t)\ \{e\}]\!]^{exp}_{s,t} & := & P(S') := s \quad \textit{(new } S'\textit{)} \\
 & \wedge & \mathcal{S}c(x_i^{\mathsf{D}}) := S' \\
 & \wedge & x_i^{\mathsf{D}} : t' \\
 & \wedge & t \equiv Fun[t', \tau_2] \quad \textit{(fresh } \tau_2\textit{)} \\
 & \wedge & C \ \wedge\ [\![e]\!]^{exp}_{S',\tau_2} \\
 & & \textit{where } [\![t]\!]^{ty}_{s} = (t', C) \\
[\![\texttt{letrec}\ bs\ \texttt{in}\ e]\!]^{exp}_{s,t} & := & P(S') := s \quad \textit{(new } S'\textit{)} \\
 & \wedge & [\![bs]\!]^{bind^*}_{S',S'} \ \wedge\ [\![e]\!]^{exp}_{S',t} \\
[\![\texttt{letpar}\ bs\ \texttt{in}\ e]\!]^{exp}_{s,t} & := & P(S') := s \quad \textit{(new } S'\textit{)} \\
 & \wedge & [\![bs]\!]^{bind^*}_{S,S'} \ \wedge\ [\![e]\!]^{exp}_{S',t} \\
[\![\texttt{Int}]\!]^{ty}_{s} & := & (Int, \mathsf{True}) \\
[\![\texttt{Bool}]\!]^{ty}_{s} & := & (Bool, \mathsf{True}) \\
[\![t_1 \to t_2]\!]^{ty}_{s} & := & (Fun[t_1', t_2'], C_1 \wedge C_2) \\
 & & \textit{where } [\![t_1]\!]^{ty}_{s} = (t_1', C_1) \\
 & & \textit{and } [\![t_2]\!]^{ty}_{s} = (t_2', C_2) \\
[\![Xs\,.\,X_i]\!]^{ty}_{s} & := & (Rec(\delta), \quad \textit{(fresh } \delta\textit{)} \\
 & & X_i^{\mathsf{R}} \mapsto \delta \wedge [\![Xs\,.\,X_i]\!]^{qid}_{s})
\end{array}
$$

Figure 13. Constraint generation for LMR.

[2] Our actual implementation operates over the abstract syntax of LMR, and is written in DynSem **??**, a declarative domain-specific language for expressing semantics; although readable, it is relatively verbose.

$$
\begin{array}{rcll}
[\![n]\!]^{exp}_{s,t} & := & t \equiv Int & \\
[\![\texttt{true}]\!]^{exp}_{s,t} & := & t \equiv Bool \qquad [\![\texttt{false}]\!]^{exp}_{s,t} \;:=\; t \equiv Bool & \\
[\![e_1 \oplus e_2]\!]^{exp}_{s,t} & := & \oplus : Fun[\tau_1, Fun[\tau_2, \tau_3]] \wedge t \equiv \tau_3 \wedge [\![e_1]\!]^{exp}_{s,\tau_1} \wedge [\![e_2]\!]^{exp}_{s,\tau_2} & \textit{(fresh } \tau_1, \tau_2, \tau_3\textit{)} \\
[\![\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3]\!]^{exp}_{s,t} & := & t \textit{ is } \tau_2 \sqcup \tau_3 \wedge [\![e_1]\!]^{exp}_{s,Bool} \wedge [\![e_2]\!]^{exp}_{s,\tau_2} \wedge [\![e_3]\!]^{exp}_{s,\tau_3} & \textit{(fresh } \tau_2, \tau_3\textit{)} \\
[\![Xs\,.\,x_i]\!]^{exp}_{s,t} & := & x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : t \wedge [\![Xs\,.\,x_i]\!]^{qid}_{s} & \textit{(fresh } \delta\textit{)} \\
[\![e_1\ e_2]\!]^{exp}_{s,t} & := & \tau_2 \preceq \tau_1 \wedge [\![e_1]\!]^{exp}_{s,Fun[\tau_1,t]} \wedge [\![e_2]\!]^{exp}_{s,\tau_2} & \textit{(fresh } \tau_1, \tau_2\textit{)} \\
[\![e\,.\,x_i]\!]^{exp}_{s,t} & := & P(S') := \bot \wedge \varsigma \in \mathcal{IS}(S') \wedge \mathcal{S}c(x_i^{\mathsf{R}}) := S' \wedge \delta_1 \rightsquigarrow \varsigma \wedge x_i^{\mathsf{R}} \mapsto \delta_2 & \textit{(new } S'\textit{)(fresh } \delta_1, \delta_2, \varsigma\textit{)} \\
 & \wedge & \delta_2 : t \wedge [\![e]\!]^{exp}_{s,Rec(\delta_1)} & \\
[\![\texttt{with } e_1 \texttt{ do } e_2]\!]^{exp}_{s,t} & := & P(S') := s \wedge \varsigma \in \mathcal{IS}(S') \wedge \delta \rightsquigarrow \varsigma \wedge [\![e_1]\!]^{exp}_{s,Rec(\delta)} \wedge [\![e_2]\!]^{exp}_{S',t} & \textit{(new } S'\textit{)(fresh } \delta, \varsigma\textit{)} \\
[\![\texttt{new } Xs\,.\,X_i\ \{\,bs\,\}]\!]^{exp}_{s,t} & := & P(S') := s \wedge X_i^{\mathsf{R}} \in \mathcal{I}(S') \wedge X_i^{\mathsf{R}} \mapsto \delta \wedge t \equiv Rec(\delta) & \textit{(new } S'\textit{)(fresh } \delta\textit{)} \\
 & \wedge & [\![Xs\,.\,X_i]\!]^{qid}_{s} \wedge [\![bs]\!]^{fbind^*}_{s,s'} & \\
[\![x_i = e]\!]^{fbind}_{s_r,s_d} & := & \mathcal{S}c(x_i^{\mathsf{R}}) := s_r \wedge x_i^{\mathsf{R}} \mapsto \delta \wedge \delta : \tau_1 \wedge \tau_2 \preceq \tau_1 \wedge [\![e]\!]^{exp}_{s_r,\tau_2} & \textit{(fresh } \delta, \tau_1, \tau_2\textit{)}
\end{array}
$$

Figure 14. Constraint generation for LMR.

- A **`module`** builds a new lexical child scope for its declarations, so the generator function creates a new scope S' and generates assumptions that s is the parent of S' and that the module name declaration is associated with S'. It then conjoins the constraints obtained by recursively invoking $[\![-]\!]^{decl}_{S'}$ on its member declarations.
- An **`import`** doesn't declare an identifier, but instead generates constraints forcing the imported module name into the import set and reference set for s. The $[\![-]\!]^{qid}_{s}$ invocation generates additional constraints needed to describe references to potentially qualified names. (We omit discussion of the details, which are slightly complex.)
- A **`def`** invokes an auxiliary generating function $[\![-]\!]^{bind}_{s_r,s_d}$ to process the definition; s_d is the scope into which the defined identifier's declaration should be installed, and s_r is the scope into which any identifier references in the defininging expression should go. (In this invocation, the two scope parameters are the same, but the *bind* generator is invoked at other places, e.g. for the **`letpar`** expression, where they are not.) For bindings without explicit type annotation, a constraint is generated giving the defined identifier a fresh type τ, and function $[\![-]\!]^{exp}_{s,t}$ is invoked to generate constraints for the defining expression with the expected type $t = \tau$. (We discuss generation of expression constraints below.) For bindings with an explicit concrete type annotation, we also generate a constraint that τ be a subtype of the declared type, after it is translated into an internal type constructor using auxiliary function $[\![-]\!]^{ty}_{s}$. This function, unlike all the others, returns a pair of things: the internalized type constructor, and any constraints generated by references to record names (which are installed into the scope given by parameter s).
- A **`record`** is handled similarly to a **`module`**, but the details are more complicated. If the record has a super-type (non-empty **`extends`** clause), an auxiliary function $[\![-]\!]^{sup}_{s_r,s_d,t}$ is invoked to generate constraints to describe the inheritance. Parameter $s_r = s$ is the scope in which the super-type name is to be resolved, $s_d = S'$ is the scope into which the super-type's scope is to be imported, and $t = Rec(X_i^{\mathsf{D}})$ is the new record type. The *sup* generator builds a subtyping assumption (the only source of such assumptions in LMR) that relates the new record type to the result of resolving the super-type name, via a fresh declaration variable δ. The fields of the new record are declared and given type constraints by another auxiliary function.

Constraint generation for expressions is largely straightforward. The *expr* generator is parameterized by the scope s into which references should be installed and the expected type t of the expression (often a type variable). Expressions introducing local bindings re-use the *bind* generator. (There is function for sequential **`let`**, which is desugared into nested **`letpar`** expressions before constraint generation is performed.) Note that generated constraints always force an expression to have a precise type, which is designed to be minimal in the subtyping hierarchy. Subtyping is allowed only at function applications, at bindings to explicitly annotated identifiers, and in the conditional expressions, for which a least-upper-bound constraint is generated. Scope variables ς are introduced only for field dereference and **`with`** expressions.

6. Resolution Algorithm

TODO ► *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?*◄

In this section, we describe an algorithm for computing program resolutions in the sense of Section 4.3. Suppose we have a program p from which we collect a set of assumptions $[\![p]\!] = A_p \wedge C_p$, where A_p is a conjunction of assumptions and C_p is a conjunction of pure constraints. Then recall that a *resolution* for p is a multi-sorted substitution ϕ and a typing environment ψ such that

$$|\phi(A_p)|, \psi \models \phi(C_p) \qquad (\diamond)$$

Our algorithm works only for a restricted class of generated constraints: all assumptions must be ground, except that (i) scope variables ς can appear in direct import assumptions (e.g. $\varsigma \in \mathcal{IS}(S)$), and (ii) type variables τ and declaration variables δ can appear on the right-hand side of a subtyping assumption (e.g. $Rec(A_i^{\mathsf{D}}) <: Rec(\delta)$). This restriction is met by the constraints generated by the LMR collection algorithm in Section 5. Broader classes of constraints might be useful for other languages; we defer exploration of algorithms that could handle these to future work.

6.1 Handling Variable in Assumptions

The basic approach of the algorithm is to apply the definitions in Section 4.2.1 to the assumptions to build a scope graph and a subtyping relation, and then use these to resolve pure constraints of the form $x^{\mathsf{R}} \mapsto d$ or $t_1 \preceq t_2$ in the context of a conventional unification-based algorithm. However, since the assumptions can contain variables, we cannot fully define the scope graph or subtyping relation before starting constraint resolution, because we don't fully know ϕ. Thus, our algorithm builds ϕ (and Ψ) incrementally. The key idea is that we can resolve some pure constraints even when ϕ is not yet fully defined, in such a way that the resolution remains valid as it becomes more defined.

Sub-typing The construction of the sub-typing relation from a set of ground assumptions given in Section 4.2.1 is monotonic. Let $\leq_A$ be the subtyping order generated from a set of ground assumptions A. Then given two sets of grounds assumptions A_1 and A_2, we have the following property:

$$A_1 \subseteq A_2 \Rightarrow T_1 \leq_{A_1} T_2 \Rightarrow T_1 \leq_{A_2} T_2$$

If A is any set of (not necessarily ground) assumptions, and $\overline{A}$ is its subset of ground assumptions, then for all substitutions ϕ mapping type variable to ground types we have:

$$T_1 \leq_{\overline{A}} T_2 \Rightarrow T_1 \leq_{\phi(A)} T_2$$

If we can deduce a subtyping relation between two types by only using the ground assumptions then this relation will still hold under any subsequent substitution. Therefore, if at some point of the resolution we have the partial substitution ϕ, meaning that if there is a solution ϕ_p then there exists σ such that $\phi_p = \sigma \circ \phi$, then every sub-typing we can deduce at this point using the assumptions grounded by ϕ will remain true in the final solution:

$$T_1 \leq_{\overline{\phi(A_p)}} T_2 \Rightarrow T_1 \leq_{\phi'(A_p)} T_2$$

APT ►*This last equation uses undefined things. (And could just omit.)*◄

Scope Graphs The situation is a bit more complicated with respect to scope graphs. The non-strictly positive premise of the (V) rule of the resolution calculus makes the derivation of a resolution relation from a graph non-monotonic with respect to additions to the graph. For example, suppose that in some graph $\mathcal{G}$ a reference x^{R} in a scope S resolves to declaration x_i^{D} in the parent scope S'. In a bigger graph $\mathcal{G}'$ that also has a declaration $x_{i'}^{\mathsf{D}}$ in S itself, x^{R} will resolve to $x_{i'}^{\mathsf{D}}$, and the old resolution to x_i^{D} will be shadowed. Therefore we can not simply resolve a reference in a graph built from ground assumptions and expect this resolution to remain valid later in the resolution process.

However, we have restricted the set of constraints we handle so that almost all assumptions used for scope graph construction are in fact ground from the beginning. The only exception is for direct import declarations, where the imported scope can be a scope variable; this construction is essential for expressing record field access, where the resolution of the field name depends on the type of the record expression. In order to handle these unknown direct imports, we define an extension of the scope graph structure, called an *incomplete scope graph*, that also allows scope variables as direct imports in addition to ground scopes. The construction of the incomplete scope graph from a set of assumptions with variable direct imports is identical to the one for ordinary scope graph construction given in Section 4.2.1.

The resolution calculus as presented in Fig. 10 is only defined on ground scope graphs. Given an incomplete scope graph $\mathcal{G}$, a reference x^{R} is said to resolve to a declaration x_i^{D} if and only if this resolution is valid in all ground instance of this incomplete graph:

$$\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \triangleq \forall\, \sigma,\, \vdash_{\mathcal{G}.\sigma} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \qquad (\blacklozenge)$$

where we write $\vdash_{\mathcal{G}}$ for the resolution relation for graph $\mathcal{G}$ and $\mathcal{G}.\sigma$ is the ground scope graph corresponding to the application of substitution σ to variables in $\mathcal{G}$.

In order to be able to detect eventual duplicate resolutions in the program we also want to ensure that an incomplete graph provides *all* the possible resolutions of a given reference. In particular, if a resolution is unique in an incomplete graph, we want it to be unique in all its ground instances. An incomplete graph $\mathcal{G}$ is stable for a reference x^{R}, denoted

$\mathcal{G} \uparrow x^{\mathsf{R}}$, if all the resolutions in all its ground instances are the same:

$$\mathcal{G} \uparrow x^{\mathsf{R}} \triangleq \forall \sigma, \sigma' \vdash_{\mathcal{G}.\sigma} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \Rightarrow \vdash_{\mathcal{G}.\sigma'} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$$

The resolution algorithm in Fig. 15 defines resolution in (potentially) incomplete scope graphs. TODO ▶*Define ◁ operator.*◀ This algorithm raises an exception if the graph is not stable for the reference.

$$x_i^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) \Longrightarrow \vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}} \wedge \mathcal{G} \uparrow x^{\mathsf{R}} \qquad (\spadesuit)$$

where $R_{\mathcal{G}}(x^{\mathsf{R}})$ denotes the main resolution function $R[\emptyset](x^{\mathsf{R}})$ of the graph $\mathcal{G}$.

We now sketch of proof of correctness for this algorithm. First, notice that the algorithm terminates using the lexicographic ordering $(\#(\mathcal{R}(\mathcal{G})\backslash\mathbb{I}), \#(\mathcal{S}(G)\backslash\mathbb{S}))$, where $\#(A)$ denotes the cardinality of set A. We next prove that on ground scope graphs, this algorithm behaves like the standard resolution algorithm presented in [9]. If $\mathcal{G}$ is ground then:

$$R_{\mathcal{G}}[\mathbb{I}](x^{\mathsf{R}}) = \{x_i^{\mathsf{D}} \mid \mathbb{I} \vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}\} \qquad \text{(i)}$$

Proof. In this case, since the graph is ground, no exceptions can be thrown. Therefore the proof is an adaptation of Theorem 1 of [9]. The only differences are: (a) the extra case of direct imports, which can be simply handled by adapting the name import case of the original proof; and (b) the fact that instead of computing the complete sets of visible and reachable declarations, the auxiliary algorithms only compute the ones matching the name argument. □

Now let $\mathcal{G}$ be an incomplete scope graph and $\mathcal{G}'$ one of its instances. If a resolution on $\mathcal{G}$ terminates with a set of declarations then the resolution on $\mathcal{G}'$ does too:

$$R_{\mathcal{G}}[\mathbb{I}](x^{\mathsf{R}}) = S \Longrightarrow R_{\mathcal{G}'}[\mathbb{I}](x^{\mathsf{R}}) = S \qquad \text{(ii)}$$

Proof. By induction on the termination order of the algorithm $(\#(\mathcal{R}(()\mathcal{G})\backslash\mathbb{I}), \#(\mathcal{S}(G)\backslash\mathbb{S}))$. Since exceptions are never caught, and since an exception is triggered as soon as a scope variable is encountered, if the a run of the algorithm on $\mathcal{G}$ starting from x^{R} does terminate with a with a result then this run is exactly the same on $\mathcal{G}'$. □

Finally, we can prove ♠:

Proof. Let $S = R_{\mathcal{G}}(x^{\mathsf{R}})$ and pick $x_i^{\mathsf{D}} \in S$.

- To prove that x^{R} resolves to x_i^{D} in $\mathcal{G}$, let $\mathcal{G}'$ be an arbitrary ground instance of $\mathcal{G}$. Using (ii) we have $x_i^{\mathsf{D}} \in R_{\mathcal{G}'}(x^{\mathsf{R}})$ and by (i) we have $\vdash_{\mathcal{G}'} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$. By ♦, we get that $\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x_i^{\mathsf{D}}$.
- To prove stability, let $\mathcal{G}_1$ and $\mathcal{G}_2$ be ground instances of $\mathcal{G}$. Then by (ii), $R_{\mathcal{G}_1}(x^{\mathsf{R}}) = S = R_{\mathcal{G}_2}(x^{\mathsf{R}})$, so by definition we have $\mathcal{G} \uparrow x^{\mathsf{R}}$.

□

6.2 Constraint solving algorithm

In Fig. 6.2 we present an algorithm to solve the constraint system from Section 4. The algorithm is a non-deterministic rewrite system of a tuple $(C, \mathcal{G}, <:, `\psi)$ of a constraint, a scopegrah, a sub-typing relation and a typing environment. It is non-deterministic in the sense that rules may be applied at any time and constraint clauses can be matched in any order.

Name resolution introduces ambiguity, since a reference x^{R} may resolve to multiple definitions. If this happens the solver branches, picking a different resolution for x^{R} in every branch. The returned solution is a set of all the $(C, \mathcal{G}, <:, \psi)$ tuples the solver was able to construct.

The initial state of the solver is the collected constraint, the (incomplete) scopegraph and sub-typing relation from the assumptions and an empty typing environment. The algorithm will eliminate clauses from C while instantiating $\mathcal{G}$ and $<:$ and filling ψ. The algorithm terminates when the constraint is empty or no more clauses can be solved. Any unsolved constraint corresponds to some error in the program.

All the rules solve one constraint, possibly updating components of the tuple or applying a substitution to it. The S-RESOLVE rule solves $x^{\mathsf{R}} \mapsto \delta$ constraints using the resolution algorithm from Fig. 15. If a resolution is found, it is substituted for the variable δ. If the scopegraph is incomplete, the algorithm might throws an exception, in which case the constraint is left to to be solved later. HvA ▶*What do we say about rejecting non-singleton resolutions?*◀

The S-ASSOC rule solves $x^{\mathsf{D}} \rightsquigarrow \varsigma$ constraints, by looking up ground declaration x^{D}'s associated scope S in the scope graph. By subtituting S for ς, the scopegraph becomes more complete, possible allowing more references to be resolved.

Type equality constrains $T_1 \equiv T_2$ are solved by the S-EQUAL rule. It uses first order unification, as described in [1]. The resulting substitution is applied to the tuple.

Rule S-SUBTYPE solves constraints of the form $t_1 \preceq t_2$ by checking that $t_1 \leq_{<:} t_2$ for the ground types t_1 and t_2. The check might not succeed of $<:$ is incomplete, in which case it might be solved later.

Rule S-LUB solves T *is* $t_1 \sqcup t_2$ constraints. It does so by calculating the least upper bound $t = (t_1 \sqcup t_2)$ of the ground types t_1 and t_2 and generating a new equality constraint $T \equiv t$. The solver dependends here on a language specific leat upper bound function $\sqcup$, which for LMRis presented in Fig. 16.

Constraints of the form $x^{\mathsf{D}} : T$ are solved by rules S-DECLTYPEFIRST and S-DECLTYPENEXT. The first rule is used the first time x^{D} is encountered and just adds it to the typing environment. For every next encounter, the other rule unifies the type T from the constraint with the type $\psi(x^{\mathsf{D}})$ from the typing environment. The resuling substitution is applied to the tuple.

The trivial constraint True is handled by S-TRUE.

$$
\begin{aligned}
R[\mathbb{I}](x^{\mathsf{R}}) &:= R_V[\{x^{\mathsf{R}}\} \cup \mathbb{I}, \{\}](x, \mathcal{S}c(x^{\mathsf{R}})) \\
R_V[\mathbb{I},\mathbb{S}](x,S) &:= R_L[\mathbb{I},\mathbb{S}](x,S) \lhd R_P[\mathbb{I},\mathbb{S}](x,S) \\
R_L[\mathbb{I},\mathbb{S}](x,S) &:= R_D[\mathbb{I},\mathbb{S}](x,S) \lhd R_I[\mathbb{I},\mathbb{S}](x,S) \\
R_D[\mathbb{I},\mathbb{S}](x,S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \{x_i^{\mathsf{D}} | x_i^{\mathsf{D}} \in \mathcal{D}(S)\} \end{cases}
\end{aligned}
\qquad
\begin{aligned}
R_I[\mathbb{I},\mathbb{S}](x,S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ \textbf{\textit{exception}} \text{ if } \mathcal{IS}(S) \text{ contains a variable} \\ \bigcup \left\{ R_L[\mathbb{I},\{S\} \cup \mathbb{S}](x,S') \;\middle|\; S' \in IS[\mathbb{I}](S) \cup \mathcal{IS}(S) \right\} \end{cases} \\
IS[\mathbb{I}](S) &:= \{\mathcal{DS}c(y^{\mathsf{D}}) \mid y^{\mathsf{R}} \in \mathcal{I}(S) \backslash \mathbb{I} \wedge y^{\mathsf{D}} \in R[\mathbb{I}](y^{\mathsf{R}})\} \\
R_P[\mathbb{I},\mathbb{S}](x,S) &:= \begin{cases} \emptyset \text{ if } S \in \mathbb{S} \\ R_V[\mathbb{I},\{S\} \cup \mathbb{S}](x, \mathcal{P}(S)) \end{cases}
\end{aligned}
$$

Figure 15. Name resolution algorithm

$$
\begin{aligned}
(\mathit{Int} \sqcup \mathit{Int}) &\xrightarrow{A} \mathit{Int} && \text{(LUB-INT)} \\
(\mathit{Rec}(d) \sqcup \mathit{Rec}(d')) &\xrightarrow{A} \begin{cases} \mathit{Rec}(d) & \text{if } \mathit{Rec}(d') \leq_A \mathit{Rec}(d) \\ (\mathit{Rec}(d) \sqcup \mathit{Rec}(d'')) & \text{if } \mathit{Rec}(d') <: \mathit{Rec}(d'') \in A \end{cases} && \text{(LUB-REC)} \\
(\mathit{Fun}[t_1,t_2] \sqcup \mathit{Fun}[t_3,t_4]) &\xrightarrow{A} \mathit{Fun}[(t_1 \sqcap t_3),(t_2 \sqcup t_4)] && \text{(LUB-ARROW)} \\
(\mathit{Int} \sqcap \mathit{Int}) &\xrightarrow{A} \mathit{Int} && \text{(GLB-INT)} \\
(\mathit{Rec}(d) \sqcap \mathit{Rec}(d')) &\xrightarrow{A} \begin{cases} \mathit{Rec}(d) & \text{if } \mathit{Rec}(d) \leq_A \mathit{Rec}(d') \\ \mathit{Rec}(d') & \text{if } \mathit{Rec}(d') \leq_A \mathit{Rec}(d) \end{cases} && \text{(GLB-REC)} \\
(\mathit{Fun}[t_1,t_2] \sqcap \mathit{Fun}[t_3,t_4]) &\xrightarrow{A} \mathit{Fun}[(t_1 \sqcup t_3),(t_2 \sqcap t_4)] && \text{(GLB-ARROW)}
\end{aligned}
$$

Figure 16. LMR specific functions

$$
\begin{aligned}
(x^{\mathsf{R}} \mapsto \delta \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, <:, \psi) && \text{(S-RESOLVE)} \\
&\quad \textit{where } x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}}) \textit{ without exception} \\
(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow [\varsigma \mapsto S](C, \mathcal{G}, <:, \psi) && \text{(S-ASSOC)} \\
&\quad \textit{where } \mathcal{DS}c(x^{\mathsf{D}}) = S \\
(T_1 \equiv T_2 \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow \sigma(C, \mathcal{G}, <:, \psi) && \text{(S-EQUAL)} \\
&\quad \textit{where } \mathcal{U}(T_1, T_2) \longrightarrow \sigma \\
(T \textit{ is } t_1 \sqcup t_2 \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow (T \equiv t \wedge C, \mathcal{G}, <:, \psi) && \text{(S-LUB)} \\
&\quad \textit{where } <: \textit{ is ground and } (t_1 \sqcup t_2) \xrightarrow{<:} t \\
(t_1 \preceq t_2 \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow (C, \mathcal{G}, <:, \psi) && \text{(S-SUBTYPE)} \\
&\quad \textit{where } t_1 \leq_{<:} t_2 \\
(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow \begin{cases} (C, \mathcal{G}, <:, \{x^{\mathsf{D}} \mapsto T\} \cup \psi) & \text{if } x^{\mathsf{D}} \notin \mathrm{dom}(\psi) \\ (\psi(x^{\mathsf{D}}) \equiv T \wedge C, \mathcal{G}, <:, \psi) & \text{else} \end{cases} && \text{(S-TYPEOF)} \\
(\mathsf{True} \wedge C, \mathcal{G}, <:, \psi) &\longrightarrow (C, \mathcal{G}, <:, \psi) && \text{(S-TRUE)}
\end{aligned}
$$

Figure 17. Constraint solving algorithm

6.3 Verification

We want to prove the soundness of the constraint resolution algorithm, that is, that the solver produces a correct solution to the program resolution problem. If the solver reduces to an empty set of constraints, then the initial constraint was satisfiable. Moreover we want to ensure that the produced typing environment is a valid one, that is, it corresponds to a solution. Therefore we want to ensure the following

property:

$$\forall C, \mathcal{G}, <:, \Psi, \mathcal{G}_2, <:_2, \Psi_2,$$
$$(C, \mathcal{G}, <:, \Psi) \longrightarrow^* (\mathsf{True}, \mathcal{G}', <:', \Psi') \Rightarrow$$
$$\exists \sigma, \sigma(\mathcal{G}, <:), \Psi' \models \sigma(C_1) \quad (\diamondsuit)$$

Proof. In order to prove this result we first state some results about the auxiliary unification and least upper bound computations.

Unification The unification of two terms t_1 and t_2 aims at producing a substitution σ such that if $\mathcal{U}(t_1, t_2) = \sigma$ then:

$$\sigma t_1 = \sigma t_2 \wedge \sigma\sigma = \sigma$$

See [1] for a survey on unification problem and unification algorithms for first order terms.

Least Upper Bound The least upper bound of a set of terms S for a given order $\leq$ is a term t, denoted $\sqcup_{\leq} T$ such that:

$$\forall s \in S, s \leq t \wedge \forall t', \forall s \in S, s \leq t' \Rightarrow t \leq t'$$

Given a set of ground subtyping assumptions A of the form $Rec(A) <: Rec(B)$, such that a type T appears only once on the left hand side of an assumption, then the reduction $\xrightarrow{A}$ presented in Figure **??** compute the least upper bound. Given the restriction on the set of assumptions due to unique inheritance, this computation is similar the computation of common ancestors of two nodes in a tree.

Resolution Soundness We now can prove the property $\diamondsuit$ of the constraint resolution algorithm. We first prove that for each reduction step, if the output is satisfiable the input is also satisfiable in the same definition to type environment. This is stated by the following property:

$$\forall (C_1, \mathcal{G}_1, <:_1, \Psi_1), (C_2, \mathcal{G}_2, <:_2, \Psi_2),$$
$$(C_1, \mathcal{G}_1, <:_1, \Psi_1) \longrightarrow (C_2, \mathcal{G}_2, <:_2, \Psi_2) \Rightarrow$$
$$\forall \sigma, \sigma(\mathcal{G}_2, <:_2, \Psi_2) \models \sigma(C_2) \Rightarrow$$
$$\exists \sigma', \sigma'(\mathcal{G}_1, <:_1, \Psi_2) \models \sigma'(C_1) \quad (\dagger)$$

The proof of this property is done by case analysis on the reduction step:

$$(C_1, \mathcal{G}_1, <:_1, \Psi_1) \longrightarrow (C_2, \mathcal{G}_2, <:_2, \Psi_2)$$

and is presented in Appendix A.1.

Using this result †, we can prove property $\diamondsuit$ by a simple induction on the number of reduction steps.

□

7. Validation

LMR and argue why that is representative

8. Related Work

Interaction between Names and Types anything?
proposal for type-directed name resolution in Haskell
https://ghc.haskell.org/trac/haskell-prime/wiki/TypeDirectedNameReso

Formalisms for Type Systems TinkerType [6]
Ruler [3]
Typical: Taking the Tedium Out of Typing [5]
Ott [12]
Lem [8]
A generator for type checkers [4]

Type Inference Algorithms A Theory of Type Polymorphism in Programming [7]
A Simple Algorithm and Proof for Type Inference [16]
Pierce Local type inference [11]
Pierce Advanced [10]
HM(X) is CLP [14]
Efficient local type inference [2]
OutsideIn(X) Modular type inference with local assumptions [15]

Applications future work: a generic framework for refactoring following the approach of constraint-based refactoring [13]

9. Conclusion

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[2] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni. Efficient local type inference. In G. E. Harris, editor, *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 475–492. ACM, 2008.

[3] A. Dijkstra and S. D. Swierstra. Ruler: Programming type rules. In M. Hagiya and P. Wadler, editors, *Functional and Logic Programming, 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006, Proceedings*, volume 3945 of *Lecture Notes in Computer Science*, pages 30–46. Springer, 2006.

[4] H. Gast. *A Generator for Type Checkers*. PhD thesis, Eberhard-Karls-UniversitÃd't, TÃijbingen, 2004.

[5] R. Grimm, L. Harris, and A. Le. Typical: Taking the tedium out of typing. NYU CS Technical Report TR2007-904, New York University, New York, November 2007.

[6] M. Y. Levin and B. C. Pierce. Tinkertype: a language for playing with formal systems. *Journal of Functional Programming*, 13(2):295–316, 2003.

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

[8] D. P. Mulligan, S. Owens, K. E. Gray, T. Ridge, and P. Sewell. Lem: reusable engineering of real-world semantics. In J. Jeuring and M. M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 175–188. ACM, 2014.

[9] 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.

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

[11] B. C. Pierce and D. N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, 2000.

[12] P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strnisa. Ott: effective tool support for the working semanticist. In R. Hinze and N. Ramsey, editors, *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 1–12. ACM, 2007.

[13] F. Steimann and J. von Pilgrim. Constraint-based refactoring with foresight. In J. Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 535–559. Springer, 2012.

[14] M. Sulzmann and P. J. Stuckey. Hm(x) type inference is clp(x) solving. *Journal of Functional Programming*, 18(2):251–283, 2008.

[15] D. Vytiniotis, S. L. P. Jones, T. Schrijvers, and M. Sulzmann. Outsidein(x) modular type inference with local assumptions. *J. Funct. Program.*, 21(4-5):333–412, 2011.

[16] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Infomaticae*, 10:115–122, 1987.

A. Proofs

A.1 Proof of property † in Section 6.3

We want to prove the following property about the constraint resolution system presented in Figure 17:

$$\forall(C_1, \mathcal{G}_1, <:_1, \Psi_1), (C_2, \mathcal{G}_2, <:_2, \Psi_2),$$
$$(C_1, \mathcal{G}_1, <:_1, \Psi_1) \longrightarrow (C_2, \mathcal{G}_2, <:_2, \Psi_2) \Rightarrow$$
$$\forall\sigma, \sigma(\mathcal{G}_2, <:_2, \Psi_2) \models \sigma(C_2) \Rightarrow$$
$$\exists\sigma', \sigma'(\mathcal{G}_1, <:_1, \Psi_2) \models \sigma'(C_1) \quad (\dagger)$$

Proof. We prove this property by case analysis on the reduction:

$$(C_1, \mathcal{G}_1, <:_1, \Psi_1) \longrightarrow (C_2, \mathcal{G}_2, <:_2, \Psi_2)$$

S-RESOLVE Assume:

$$(x^{\mathsf{R}} \mapsto \delta \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow [\delta \mapsto x^{\mathsf{D}}](C, \mathcal{G}, <:, \Psi)$$

where $x^{\mathsf{D}} \in R_{\mathcal{G}}(x^{\mathsf{R}})$ and let σ' be $[\delta \mapsto x^{\mathsf{D}}]$.
Assume there is σ such that

$$\sigma(\sigma'(\mathcal{G}, <:, \Psi))) \models \sigma(\sigma' C) \quad \text{(H)}$$

then we want to prove:

$$\exists\sigma_1, \sigma_1(\mathcal{G}, <:, \sigma'\Psi)) \models \sigma_1(x^{\mathsf{R}} \mapsto \delta \wedge C)$$

We have:

1. $\vdash_{\mathcal{G}} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}$ by correctness of the name resolution algorithm $R_{\mathcal{G}}()$,
2. $\vdash_{\sigma\sigma'\mathcal{G}} x^{\mathsf{R}} \longmapsto x^{\mathsf{D}}$ by definition,
3. $\sigma\sigma'(\mathcal{G}, <:, \sigma'\Psi) \models \sigma\sigma' x^{\mathsf{R}} \mapsto \delta$
4. $(\sigma\sigma')(\mathcal{G}, <:, \Psi)) \models (\sigma\sigma')C$ using H
5. $(\sigma\sigma')(\mathcal{G}, <:, \sigma'\Psi))) \models (\sigma\sigma')C$ since $\sigma'\sigma' = \sigma'$
6. we conclude with $\sigma_1 = (\sigma\sigma')$ by C-AND rule of the constraint interpretation with 3. and 5.

****

S-ASSOC Assume:

$$(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow [\varsigma \mapsto S](C, \mathcal{G}, <:, \Psi)$$

where $\mathcal{DS}c(x^{\mathsf{D}}) = S$ and let σ' be $[\varsigma \mapsto S]$.
Assume there is σ such that

$$\sigma(\sigma'(\mathcal{G}, <:, \Psi))) \models \sigma(\sigma' C) \quad \text{(H)}$$

then we want to prove:

$$\exists\sigma_1, \sigma_1(\mathcal{G}, <:, \sigma'\Psi)) \models \sigma_1(x^{\mathsf{D}} \rightsquigarrow \varsigma \wedge C)$$

We have:

1. $\mathcal{DS}c(x^{\mathsf{D}}) = S$ by the rewriting rule condition
2. $\sigma\sigma'(\mathcal{G}, <:, \sigma'\Psi) \models \sigma\sigma' x^{\mathsf{D}} \rightsquigarrow \varsigma$
3. $(\sigma\sigma')(\mathcal{G}, <:, \Psi)) \models (\sigma\sigma')C$ using H
4. $(\sigma\sigma')(\mathcal{G}, <:, \sigma'\Psi))) \models (\sigma\sigma')C$ since $\sigma'\sigma' = \sigma'$
5. we conclude with $\sigma_1 = (\sigma\sigma')$ by C-AND rule of the constraint interpretation with 2. and 4.

****

S-EQUAL Assume:

$$(T_1 \equiv T_2 \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow \sigma'(C, \mathcal{G}, <:, \Psi)$$

where $\sigma' = \mathcal{U}(T_1, T_2)$.
Assume there is σ such that

$$\sigma(\sigma'(\mathcal{G}, <:, \Psi))) \models \sigma(\sigma' C) \quad \text{(H)}$$

then we want to prove:

$$\exists\sigma_1, \sigma_1(\mathcal{G}, <:, \sigma'\Psi)) \models \sigma_1(T_1 \equiv T_2 \wedge C)$$

We have:

1. $\sigma' t_1 = \sigma' t_2$ by unification property
2. $\sigma\sigma'(\mathcal{G}, <:, \sigma'\Psi) \models \sigma\sigma' T_1 \equiv T_2$ by C-EQUAL rule and 1.
3. $(\sigma\sigma')(\mathcal{G}, <:, \Psi))) \models (\sigma\sigma')C$ using H
4. $(\sigma\sigma')(\mathcal{G}, <:, \sigma'\Psi))) \models (\sigma\sigma')C$ since $\sigma'\sigma' = \sigma'$ by unification property
5. we conclude by C-AND rule of the constraint interpretation with 2. and 4.

****

S-LUB Assume:

$$(T \; is \; t_1 \sqcup t_2 \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow (T \equiv t \wedge C, \mathcal{G}, <:, \Psi)$$

where $(t_1 \sqcup t_2) \xrightarrow{<:} t$.
Assume there is σ such that

$$\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(t \equiv T \wedge C) \quad \text{(H)}$$

then we want to prove:

$$\exists\sigma_1, \sigma_1(\mathcal{G}, <:, \Psi)) \models \sigma_1(T \; is \; t_1 \sqcup t_2 \wedge C)$$

We have:

1. $\sigma t = \sigma T$ by inversion of C-AND and C-EQUAL semantics rules on H
2. $t = \sqcup_{<:}\{t_1, t_2\}$ by correctness of $(\sqcup)$ reduction
3. $\sigma t = t \wedge \sigma t_1 = t_1 \wedge \sigma t_2 = t_2$ since these are ground terms
4. $\sigma(\mathcal{G}, <:, \Psi) \models \sigma(T \; is \; t_1 \sqcup t_2$ since $<:$ is ground
5. $\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(C)$ using H
6. we conclude by C-AND rule of the constraint interpretation with 4. and 5.

****

S-SUBTYPE Assume:

$$(t_1 \preceq t_2 \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow (C, \mathcal{G}, <:, \Psi)$$

Assume there is σ such that

$$\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(C) \qquad \text{(H)}$$

then we want to prove:

$$\exists \sigma_1, \sigma_1(\mathcal{G}, <:, \Psi)) \models \sigma_1(t_1 \preceq t_2 \wedge C)$$

We have:

1. $t_1 \leq_{\overline{<:}} t_2$ by reduction rule hypothesis
2. $\sigma t_1 \leq_{\sigma(<:)} \sigma t_2$ by $\leq_x$ monotonicity and since t_1 and t_2 are ground
3. $\sigma(\mathcal{G}, <:, \Psi) \models \sigma(t_1 \preceq t_2$ by C-Subtype semantics rule
4. we conclude by C-AND rule of the constraint interpretation with 3. and H

****

S-DECLTYPEFIRST Assume:

$$(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow (C, \mathcal{G}, <:, \{x^{\mathsf{D}} \mapsto T\} \cup \Psi)$$

Assume there is σ such that

$$\sigma(\mathcal{G}, <:, \{x^{\mathsf{D}} \mapsto T\} \cup \Psi)) \models \sigma(C) \qquad \text{(H)}$$

then we want to prove:

$$\exists \sigma_1, \sigma_1(\mathcal{G}, <:, \{x^{\mathsf{D}} \mapsto T\} \cup \Psi)) \models \sigma_1(x^{\mathsf{D}} : T \wedge C)$$

We have:

1. $\sigma(\mathcal{G}, <:, \{x^{\mathsf{D}} \mapsto T\} \cup \Psi) \models \sigma(x^{\mathsf{D}} : T)$ by C-TypeOf semantics rule
2. we conclude by C-AND rule of the constraint interpretation with 1. and H

****

S-DECLTYPENEXT Assume:

$$(x^{\mathsf{D}} : T \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow (\Psi(x^{\mathsf{D}}) \equiv T \wedge C, \mathcal{G}, <:, \Psi)$$

Assume there is σ such that

$$\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(\Psi(x^{\mathsf{D}}) \equiv T \wedge C) \qquad \text{(H)}$$

then we want to prove:

$$\exists \sigma_1, \sigma_1(\mathcal{G}, <:, \Psi)) \models \sigma_1(x^{\mathsf{D}} : T \wedge C)$$

We have:

1. $\sigma\Psi(x^{\mathsf{D}}) = \sigma T$ by inversion of C-AND and C-Equal semantics rules
2. $\sigma(\mathcal{G}, <:, \Psi) \models \sigma(x^{\mathsf{D}} : T)$ by C-TypeOf rule
3. $\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(C)$ using H
4. we conclude by C-AND rule of the constraint interpretation with 2. and 3.

****

S-TRUE Assume:

$$(\mathsf{True} \wedge C, \mathcal{G}, <:, \Psi) \longrightarrow (C, \mathcal{G}, <:, \Psi)$$

Assume there is σ such that:

$$\sigma(\mathcal{G}, <:, \Psi)) \models \sigma(C) \qquad \text{(H)}$$

then we have:

1. $\sigma(\mathcal{G}, <:, \Psi) \models \sigma\mathsf{True}$ by C-True rule
2. we conclude by C-AND rule of the constraint interpretation with 1. and H.

□