Computer Science 15-312
Type Reconstruction

Robert Harper

Spring, 1999

1 Introduction

One criticism that has been made of statically typed languages is that the type information required on programs is excessively verbose. While this may be a valid criticism of some statically-typed languages, it overlooks the possibility of automatically inferring type information from context to lessen the notational burdens of static typing. This approach to type checking was pioneered by Robin Milner in the design of ML, and has since been exploited in a number of settings, including a type system for Cobol (!) programs to detect Year 2000 problems. The purpose of this note is to explain how type inference is accomplished.

To get a feeling for what's involved, let's consider the `map` function, written in a modest extension of MinML with multi-argument functions and lists. To make things as simple as possible, the example is written using the list primitives `nil` (empty list), `hd` (head of list), `tl` (tail of list), `cons` (construct a list), and `null` (test for nil), rather than pattern-matching.

Here's `map`, written in this extension of MinML:

```
fun map (f, l) is
  if null(l) then
     nil
  else
     cons (f (hd l), map (f, (tl l)))
  fi
end
```

We may think of type reconstruction as being performed in several stages. First, we introduce *type variables* (written α, β, γ, etc) in place of the omitted type information on functions. These are intended as placeholders for type information that we will appear in the reconstructed typed term. We obtain the following *term scheme*, a pattern, or scheme, for a typed term:

```
fun map (f:α, l:β):γ is
  if null(l) then
     nil
  else
     cons (f (hd(l)), map (f, tl(l)))
  fi
end
```

The type variables α and β are placeholders for the argument types of `map`, and the type variable γ is a placeholder for the its result type.

In the next phase we examine the code of `map` to determine whether the type variables can be instantiated so as to obtain a well-typed reconstruction of the original untyped function. This is achieved by a form of type checking in which we generate *constraints* between *type schemes*. A *type scheme* is, as the name suggestions, a type expression that involves type variables acting as placeholders for types. For example, $\alpha \rightarrow \alpha\,\texttt{list}$ is a type scheme, with scheme variable α. It stands for all types of the form $\tau \rightarrow \tau\,\texttt{list}$, where τ is a type. A *constraint* is an equation between type schemes. During inspection of the code we generate constraints corresponding, for example, to the requirement that in an application, the domain type of the function must be equal to the type of the argument.

To see what's going on, let's go through the code of `map` to determine the constraints that govern its reconstruction.

1. The expression `null(l)` induces the requirement that the type β of `l` be a list type, which is expressed by the equation $\beta = \beta'\,\texttt{list}$, where β' is a "fresh" type variable (one that we haven't used before). The type of `null(l)` is `bool`.

2. The expression `nil` occurring in the "then" clause of the conditional has type $\delta\,\texttt{list}$, where δ is a fresh type variable representing the as-yet-undetermined type of the empty list.

3. The expressions `hd(l)` and `tl(l)` induce the requirements that $\beta = \beta''\,\texttt{list}$ and $\beta = \beta'''\,\texttt{list}$, where β'' and β''' are fresh type variables. The type of `hd(l)` is therefore β'' and the type of `tl(l)` is $\beta'''\,\texttt{list}$.

4. The expression `map (f, tl(l))` induces the requirements that $\alpha = \alpha$ and $\beta = \beta'''\,\texttt{list}$, which arise from equating corresponding argument and parameter types. The type of the expression is therefore γ, the result type of `map`.

5. The expression `f (hd(l))` induces the requirement that $\alpha = \beta''\to\alpha'$, where α' is a fresh type variable, corresponding to the requirement that the type of f be a function type whose domain is the type of the argument of the application. The type of `f (hd(l))` is therefore α'.

6. The expression `cons(f (hd(l)), map(f, tl(l)))` induces the requirement that γ, the type of the recursive call to `map`, be $\gamma'\,\texttt{list}$ and that the type of `f(hd(l))` be γ'. That is, we add $\gamma = \gamma'\,\texttt{list}$ and $\alpha' = \gamma'$ to the set of constraints. The type of the `cons` expression is therefore $\gamma'\,\texttt{list}$.

7. The conditional expression induces the requirement that the types of the `then` and `else` clauses are equal, that is that $\delta\,\texttt{list} = \gamma'\,\texttt{list}$.

8. Finally, the result type of the function must be the type of the body: $\gamma = \delta\,\texttt{list}$.

This completes the generation of constraints.

Here is a summary of the constraints we have generated from this program:

$$\begin{array}{rcl}
\beta & = & \beta'\,\texttt{list} \\
\beta & = & \beta''\,\texttt{list} \\
\beta & = & \beta'''\,\texttt{list} \\
\alpha & = & \alpha \\
\beta & = & \beta'''\,\texttt{list} \\
\alpha & = & \beta''\to\alpha' \\
\gamma & = & \gamma'\,\texttt{list} \\
\alpha' & = & \gamma' \\
\delta\,\texttt{list} & = & \gamma'\,\texttt{list} \\
\gamma & = & \delta\,\texttt{list}
\end{array}$$

The *solutions* to these constraints determine all of the possible reconstructions of the untyped term. If the constraints have no solution, the original expression is ill-formed and must be rejected. The determination of whether a given set of constraints has a solution and, if so, determining the set of possible solutions, is performed by a process similar to Gaussian elimination. Solving these constraints leads to the following equations

$$\begin{array}{rcl} \alpha & = & \beta' \rightarrow \gamma' \\ \beta & = & \beta' \,\texttt{list} \\ \gamma & = & \gamma' \,\texttt{list} \end{array}$$

Since the solution involves the unspecified type variables β' and γ', there are many reconstructions of `map` corresponding to how we choose to instantiate β' and γ'. That is, all reconstructions of `map` are instances of the following term scheme (renaming type variables for readability):

```
fun map (f:α→β,x:α list):β list is
  if null(l) then
     nil
  else
     cons (f (hd(l)), map (f, tl(l)))
  fi
end
```

The remainder of this note consists of a systematic development of these ideas.

2 Type Reconstruction

The syntax of MinML is *explicitly typed* (or, more briefly, *typed*) in the sense that every expression has a unique type, if it has one at all. This is accomplished by requiring the programmer to provide explicitly the argument and result types of functions. In extensions to MinML (such as the ones you will explore in homework) additional type information is often required to ensure that the uniqueness of types property is maintained.

In practice it can be quite burdensome to be forced to supply the argument and result types of functions. It would be convenient to be able to omit this information whenever it can be deduced from context. For example, we may wish to write the factorial function in the form

```
fun f(x) is if x=0 then 1 else x * f (x-1) fi end
```

rather than in the fully explicit form

```
fun f(x:int):int is if x=0 then 1 else x * f (x-1) fi end
```

The problem is even worse in cases such as the following, where the return type is a function:

```
fun f(x) is fun g(y) is x+y end end
```

In explicit form we would have to write

```
fun f(x:int):int->int is fun g(y:int):int is x+y end end
```

As programs grow larger it becomes more and more burdensome to supply the required type information.

This suggests that we introduce an *implicitly typed* (briefly, *untyped*) syntax for MinML. The untyped MinML expressions, u, are inductively-defined by the following rules:

$$\frac{}{x} \qquad \frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{u_1 \quad \ldots u_n}{o(u_1, \ldots, u_n)} \qquad \frac{u \quad u_1 \quad u_2}{\texttt{if}\, u\, \texttt{then}\, u_1\, \texttt{else}\, u_2\, \texttt{fi}}$$

$$\frac{u}{\texttt{fun}\, f\, (x)\ \texttt{is}\, u\, \texttt{end}} \qquad \frac{u_1 \quad u_2}{\texttt{apply}(u_1, u_2)}$$

The only difference is that no type information is required on functions; otherwise the syntax is the same as for MinML.

There is a simple *erasure* operation that maps a typed MinML expression e to a corresponding untyped MinML expression e^- by removing all type information. It is defined by induction on the structure of e as follows:

$$\begin{array}{rcl}
x^- & = & x \\
n^- & = & n \\
\texttt{true}^- & = & \texttt{true} \\
\texttt{false}^- & = & \texttt{false} \\
o(e_1, \ldots, e_n)^- & = & o(e_1^-, \ldots, e_n^-) \\
\texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi}^- & = & \texttt{if}\, e^-\, \texttt{then}\, e_1^-\, \texttt{else}\, e_2^-\, \texttt{fi} \\
\texttt{fun}\, f\, (x\,\texttt{:}\,\tau_1)\,\texttt{:}\,\tau_2\, \texttt{is}\, e\, \texttt{end}^- & = & \texttt{fun}\, f\, (x)\ \texttt{is}\, e^-\, \texttt{end}
\end{array}$$

It is a simple matter to check that these equations define a unique function from typed MinML to untyped MinML.

We say that the typed expression e is a *reconstruction* of the untyped expression u iff the erasure of e is u, *i.e.*, $e^- = u$. Notice that a given untyped expression may have many (in fact, infinitely many) distinct reconstructions! For example, the untyped expression `fun f(x) is x end` can be reconstructed to any typed expression of the form `fun f(x:`τ`):`τ` is x end`, where τ is a type.

How are the typed and untyped syntaxes of MinML related? We will define a four-place *elaboration* relation $\Gamma \vdash u \rightsquigarrow e : \tau$, with the intended meaning that e is a reconstruction of u with type τ, relative to the context Γ. Since a given untyped expression u may have many distinct reconstructions (or none at all, if the untyped expression is ill-formed) the elaboration relation will not be a *function* of Γ and u. The elaboration relation is inductively-defined by the following rules.

Variables reconstruct to themselves, with the type given by the context.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x \rightsquigarrow x : \tau}$$

Numbers and boolean constants reconstruct to themselves with the expected types.

$$\frac{}{\Gamma \vdash n \rightsquigarrow n : \mathtt{int}}$$

$$\frac{}{\Gamma \vdash \mathtt{true} \rightsquigarrow \mathtt{true} : \mathtt{bool}}$$

$$\frac{}{\Gamma \vdash \mathtt{false} \rightsquigarrow \mathtt{false} : \mathtt{bool}}$$

Primitive operations are reconstructed by reconstructing the constituent expressions at the required types.

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash u_n \rightsquigarrow e_n : \tau_n}{\Gamma \vdash o(u_1, \ldots, u_n) \rightsquigarrow o(e_1, \ldots, e_n) : \tau}$$

where o takes arguments of type τ_1,...,τ_n and yields values of type τ.

Conditionals reconstruct to conditionals, with the constituent expressions reconstructed at the appropriate type. Notice that we require that u reconstruct to e with type `bool`.

$$\frac{\Gamma \vdash u \rightsquigarrow e : \mathtt{bool} \qquad \Gamma \vdash u_1 \rightsquigarrow e_1 : \tau \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau}{\Gamma \vdash \mathtt{if}\, u\, \mathtt{then}\, u_1\, \mathtt{else}\, u_2\, \mathtt{fi} \rightsquigarrow \mathtt{if}\, e\, \mathtt{then}\, e_1\, \mathtt{else}\, e_2\, \mathtt{fi} : \tau}$$

Functions are reconstructed by "guessing" domain and range types, which are then assigned to the function name and parameter while reconstructing the body.

$$\frac{\Gamma[f{:}\tau_1{\rightarrow}\tau_2][x{:}\tau_1] \vdash u \rightsquigarrow e : \tau_2}{\Gamma \vdash \mathtt{fun}\, f\,(x)\;\mathtt{is}\, u\, \mathtt{end} \rightsquigarrow \mathtt{fun}\, f\;(x\,{:}\,\tau_1)\,{:}\,\tau_2\;\mathtt{is}\, e\, \mathtt{end} : \tau_1{\rightarrow}\tau_2}$$

Formally, this rule states that we may reconstruct an untyped function expression to any typed function expression with chosen domain and range types, provided that the body may be reconstructed with the specified range type, assuming the appropriate types for the function itself and its parameter.

Applications are reconstructed by "guessing" the mediating type τ_2 for the argument.

$$\frac{\Gamma \vdash u_1 \rightsquigarrow e_1 : \tau_2 {\rightarrow} \tau \qquad \Gamma \vdash u_2 \rightsquigarrow e_2 : \tau_2}{\Gamma \vdash \mathtt{apply}(u_1, u_2) \rightsquigarrow \mathtt{apply}(e_1, e_2) : \tau}$$

In other words, we may reconstruct an application expression in potentially many different ways, corresponding to the choice of the type τ_2 for the argument.

Since the elaboration rules involve arbitrary choices of type information, the elaboration relation is sometimes said to be *nondeterministic*, or *indeterminate*. This is just another way of saying that the relation is not a function of Γ and u — in general there are many possible reconstructions for given Γ and u. Although it can be helpful to think in terms of "guessing" implicit type information, it is important to realize that the elaboration relation is fully and precisely specified by these rules. There is no imprecision in the definition.

Exercise 1
Using the elaboration rules, show that the following untyped term can be reconstructed as a term of type $\mathtt{int}{\rightarrow}\mathtt{int}$. *That is, find* e *such that* $\vdash u \rightsquigarrow e : \mathtt{int}{\rightarrow}\mathtt{int}$, *where* u *is the following untyped term:*

```
(fn f in (fn x in f (f x) end) end)(fn x in x end)
```

(Here $\mathtt{fn}\,x\;\mathtt{in}\,e\;\mathtt{end}$ *stands for* $\mathtt{fun}\,f\,\mathtt{(}x\mathtt{)}\;\mathtt{is}\,e\;\mathtt{end}$, *where* $f \notin \mathrm{FV}(e)$.*) Observe the points at which the "guesses" are necessary to complete the reconstruction. How did you make the appropriate choices?*

The crucial properties of the elaboration relation are summarized in the following theorem.

Theorem 2

1. *The elaboration relation is* sound*: if* $\Gamma \vdash u \rightsquigarrow e : \tau$, *then* $\Gamma \vdash e : \tau$ *and* $e^- = u$.
2. *The elaboration relation is* complete*: if* $\Gamma \vdash e : \tau$ *and* $e^- = u$, *then* $\Gamma \vdash u \rightsquigarrow e : \tau$.

Soundness ensures that given Γ and u we can derive *only* well-typed reconstructions of u relative to Γ. Completeness ensures that we can derive *all* well-typed reconstructions of u, relative to Γ.
Proof:

1. By induction on the elaboration relation. We proceed by considering each rule in turn, applying the inductive hypothesis to the premises of the rule. We will illustrate one case here, leaving the rest as an exercise for the reader.

   Suppose that

   $$\Gamma \vdash \texttt{fun}\, f\, (x)\, \texttt{is}\, u\, \texttt{end} \leadsto \texttt{fun}\, f\ (x : \tau_1) : \tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2,$$

   where

   $$\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash u \leadsto e : \tau_2.$$

   By induction we know that

   $$\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e : \tau_2.$$

   Therefore

   $$\Gamma \vdash \texttt{fun}\, f\ (x : \tau_1) : \tau_2\, \texttt{is}\, e\, \texttt{end} : \tau_1 \to \tau_2,$$

   as required.

2. By induction on the MinML typing relation, making use of the definition of the erasure function. We will illustrate one case here, leaving the rest as an exercise for the reader.

   Suppose that $e = \texttt{fun}\, f\ (x : \tau_1) : \tau_2\, \texttt{is}\, e_2\, \texttt{end}$ and $\tau = \tau_1 \to \tau_2$, where

   $$\Gamma \vdash \texttt{fun}\, f\ (x : \tau_1) : \tau_2\, \texttt{is}\, e_2\, \texttt{end} : \tau_1 \to \tau_2,$$

   by

   $$\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e_2 : \tau_2,$$

   and that $\texttt{fun}\, f\ (x : \tau_1) : \tau_2\, \texttt{is}\, e_2\, \texttt{end}^- = u$. We are to show that $\Gamma \vdash u \leadsto e : \tau$. By the definition of the erasure function, $u = \texttt{fun}\, f\, (x)\, \texttt{is}\, u_2\, \texttt{end}$, where $u_2 = e_2^-$. Since $\tau = \tau_1 \to \tau_2$, it suffices to show that

   $$\Gamma[f{:}\tau][x{:}\tau_1] \vdash u_2 \leadsto e_2 : \tau_2,$$

   for then it follows immediately that

   $$\Gamma \vdash u \leadsto e : \tau,$$

   as required. But this follows immediately by the inductive hypothesis, since $\Gamma[f{:}\tau][x{:}\tau_1] \vdash e_2 : \tau_2$ and $u_2 = e_2^-$.

■

Exercise 3
Complete the proof of soundness and completeness of the elaboration relation.

3 Constraint Generation

The elaboration relation is a simple and elegant definition of type reconstruction for MinML, but it is not at all obvious how to implement it! The difficulty lies in the use of non-deterministic rules to "guess" omitted type information to achieve a valid reconstruction. How can these choices be made by an algorithm? The key is to defer making these choices as late as possible to take account of the constraints imposed by the context in which the expression occurs. For example, in the expression `hd(l)`, the choice of a type for `l` is constrained by the requirement that it be a list type. If this expression occurs as `hd(l)+2`, then the choice of type for `l` is further constrained to `int list` by the fact that `hd(l)` is added to the integer constant 2. Our approach to type reconstruction, then, is to first collect all of the constraints on the choices of type information by making a pass over the program, then solving these constraints to determine the missing type information. The constraints may have no solution, because the program is ill-formed, exactly one solution, because there is only one consistent set of choices for missing type information, or infinitely many solutions, because there are many ways to make consistent choices.

How do the constraints arise? Careful inspection of the elaboration rules reveals that there are implicit requirements that two types be equal in order for an expression to be well-formed. For example, the reconstruction rule for application expressions requires that the domain type of the function be equal to the argument type. Similarly, the reconstruction rule for conditionals requires that the types of the "then" and "else" clauses be equal. These conditions limit the possible choices of type information that can be used to reconstruct an untyped expression. Each of the choices is represented by an unknown whose value is constrained by a set of equations that are gleaned from an analysis of the program. By solving the for the unknowns we resolve, as late as possible, the set of choices that could be made during elaboration.

A *type scheme* is a type that may contain one or more *type variables*. We let α, β, and γ stand for type variables, and write τ^* and σ^* for type schemes (to clearly distinguish them from types, which do not have type variables). Formally, the set of type schemes is inductively defined by the following rules:

$$\frac{}{\alpha} \qquad \frac{}{\texttt{int}} \qquad \frac{}{\texttt{bool}} \qquad \frac{\tau_1^* \quad \tau_2^*}{\tau_1^* \to \tau_2^*}$$

The set $\mathrm{TV}(\tau^*)$ is the set of type variables occurring in the type scheme

τ^*. A *term scheme* is a MinML term built up using type schemes (rather than just types). We write e^* for term schemes, to emphasize that they may involve type variables. Similarly, we write Γ^* for contexts that map variables to type schemes.

A *constraint*, ϕ, is a formula expressing equations between type schemes. The set of constraints is inductively defined by the following rules:

$$\frac{}{\top} \qquad \frac{}{\tau_1^*{=}\tau_2^*} \qquad \frac{\phi_1 \quad \phi_2}{\phi_1 \wedge \phi_2}$$

The *constraint generation* relation is written $\Gamma^* \vdash u \leadsto e^* : \tau^* \; [\phi]$. The intended meaning is that in context Γ^* the untyped term u reconstructs to term scheme e^* with type scheme τ^*, subject to the constraints ϕ. (The exact meaning will be given by Theorem 4 below.) This relation will be *total* function of Γ^* and u, provided that $\mathrm{FV}(u) \subseteq \mathrm{dom}(\Gamma)$ — that is, for every Γ^* and u there will be a *unique* e^* and τ^* such that $\Gamma^* \vdash u \leadsto e^* : \tau^* \; [\phi]$, even if u is ill-formed.

The constraint generation relation is inductively-defined by the following rules.

Variables and constants are straightforward; no constraints are generated.

$$\frac{}{\Gamma^* \vdash x \leadsto x : \Gamma^*(x) \; [\top]}$$

$$\frac{}{\Gamma^* \vdash n \leadsto n : \texttt{int} \; [\top]}$$

$$\frac{}{\Gamma^* \vdash \texttt{true} \leadsto n : \texttt{bool} \; [\top]}$$

$$\frac{}{\Gamma^* \vdash \texttt{false} \leadsto n : \texttt{bool} \; [\top]}$$

Primitive operations generate constraints corresponding to the types of the arguments and results of the operation. We assume that o has argument types $\tau_1, \ldots, \tau_n$ and result type τ.

$$\frac{\Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^* \; [\phi_1] \quad \ldots \quad \Gamma^* \vdash u_n \leadsto e_n^* : \tau_n^* \; [\phi_n]}{\Gamma^* \vdash o(u_1, \ldots, u_n) \leadsto o(e_1^*, \ldots, e_n^*) : \tau \; [\phi_1 \wedge \cdots \wedge \phi_n \wedge \tau_1^* = \tau_1 \wedge \cdots \wedge \tau_n^* = \tau_n]}$$

Conditionals generate constraints for the sub-expressions, together with the requirement that the type of the test expression be `bool` and that the "then" and "else" clauses have the same type. The type variable α is uniquely associated with this occurrence of the conditional; it may not otherwise appear in a constraint.

$$\frac{\Gamma^* \vdash u \leadsto e^* : \tau^* \; [\phi] \qquad \Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^* \; [\phi_1] \qquad \Gamma^* \vdash u_2 \leadsto e_2^* : \tau_2^* \; [\phi_2]}{\begin{array}{l}\Gamma^* \vdash \texttt{if}\, u \,\texttt{then}\, u_1 \,\texttt{else}\, u_2 \,\texttt{fi} \leadsto \texttt{if}\, e^* \,\texttt{then}\, e_1^* \,\texttt{else}\, e_2^* \,\texttt{fi} : \alpha \\ \qquad\qquad\qquad\qquad [\phi \wedge \phi_1 \wedge \phi_2 \wedge \tau^*{=}\texttt{bool} \wedge \alpha{=}\tau_1^* \wedge \alpha{=}\tau_2^*]\end{array}}$$

Functions are handled by introducing fresh type variables to stand for the type of the parameter and result of the function; these variables are uniquely associated with this occurrence of the function. Constraints are then generated by examining the body of the function.

$$\frac{\Gamma^*[f{:}\alpha_1{\to}\alpha_2][x{:}\alpha_1] \vdash u \leadsto e^* : \tau^*\ [\phi]}{\Gamma^* \vdash \texttt{fun}\, f\,(x)\ \texttt{is}\, e\, \texttt{end} \leadsto \texttt{fun}\, f\ (x : \alpha_1) : \alpha_2\ \texttt{is}\, u\, \texttt{end} : \alpha_1{\to}\alpha_2\ [\phi \wedge \tau^*{=}\alpha_2]}$$

Applications are handled by introducing a fresh type variable to stand for the result type of the application.

$$\frac{\Gamma^* \vdash u_1 \leadsto e_1^* : \tau_1^*\ [\phi_1] \qquad \Gamma^* \vdash u_2 \leadsto e_2^* : \tau_2^*\ [\phi_2]}{\Gamma^* \vdash \texttt{apply}(u_1, u_2) \leadsto \texttt{apply}(e_1^*, e_2^*) : \alpha\ [\phi_1 \wedge \phi_2 \wedge \tau_1^*{=}\tau_2^*{\to}\alpha]}$$

To state precisely the properties of the constraint generation algorithm, it is necessary to introduce the notion of a *solution* to a constraint, a choice of bindings for the type variables in the constraint making all of the equations true. For example, replacing α and β with the type `int` makes the equation $\alpha{\to}\beta{=}\texttt{int}{\to}\texttt{int}$ true; it is a solution of this constraint.

A *substitution* is a function mapping type variables to type schemes that is identity almost everywhere. This means that a function S mapping type variables to type schemes is a substitution iff $S(\alpha) = \alpha$ for all but finitely many type variables α. The *domain* $\mathrm{dom}(S)$ of the substitution S is defined by $\mathrm{dom}(S) = \{\, \alpha \mid S(\alpha) \neq \alpha \,\}$. It is the set of type variables on which S is not the identity. A substitution S is *idempotent* iff $S \circ S = S$ — that is, $S(\alpha) = \hat{S}(S(\alpha))$ for all type variables α. This means that the type variables occurring in the range of S are not also in its domain. The substitution S such that $S(\alpha) = \beta{\to}\texttt{int}$ and $S(\beta) = \texttt{bool}$ is not idempotent, since $\hat{S}(S(\alpha)) = \texttt{bool}{\to}\texttt{int} \neq \beta{\to}\texttt{int} = S(\alpha)$.

The application of a substitution, S, to a type scheme, τ^*, written $\hat{S}(\tau^*)$ is defined by induction on the structure of τ^* as follows:

$$\begin{array}{rcl} \hat{S}(\alpha) & = & S(\alpha) \\ \hat{S}(\texttt{int}) & = & \texttt{int} \\ \hat{S}(\texttt{bool}) & = & \texttt{bool} \\ \hat{S}(\tau_1^*{\to}\tau_2^*) & = & \hat{S}(\tau_1^*){\to}\hat{S}(\tau_2^*) \end{array}$$

In other words, we replace all of the type variables in τ^* by their bindings in S. The type scheme $\hat{S}\tau^*$ is called an *instance* of the type scheme τ^*. Abusing notation, we write $\hat{S}(\Gamma^*)$ for the result of applying S to every type scheme in Γ^*. By a similar abuse, we also write $\hat{S}(e)$ for the application of a substitution S to a term scheme e, replacing all type variables by their bindings in S. We leave the precise definition to the reader.

Substitutions may be composed by defining $(S_1 \circ S_2)(\alpha) = \hat{S}_1(S_2(\alpha))$. That is, we first retrieve the binding of α from S_2, then apply S_1 to the resulting type scheme. The *identity* substitution, I, is defined by $I(\alpha) = \alpha$ for every type variable α. The *singleton* substitution $[\alpha{=}\tau^*]$ sends α to τ^* and is the identity elsewhere. We will often use singletons in combination with composition. The substitution $[\alpha{=}\tau^*] \circ S$ maps α to $[\tau^*/\alpha]S(\alpha)$. That is, it substitutes τ^* for α in every binding in S.

A substitution S is a *solution* for a constraint ϕ, written $S \models \phi$, every equation in ϕ is true when its type variables are instantiated by S. More precisely,

$$\begin{array}{rll} S \models \top & \textit{iff} & \textit{always} \\ S \models \tau_1^*{=}\tau_2^* & \textit{iff} & \hat{S}(\tau_1^*) = \hat{S}(\tau_2^*) \\ S \models \phi_1 \wedge \phi_2 & \textit{iff} & S \models \phi_1 \textit{ and } S \models \phi_2 \end{array}$$

We may now state the key properties of the constraint generation relation.

Theorem 4

1. *Constraint generation is a total function: For every Γ^* and u there exists a unique e^*, τ^*, and ϕ such that $\Gamma^* \vdash u \leadsto e^* : \tau^*\ [\phi]$.*

2. *Constraint generation is sound: If $\Gamma^* \vdash u \leadsto e^* : \tau^*\ [\phi]$ and $S \models \phi$, then $\hat{S}\Gamma^* \vdash u \leadsto \hat{S}e^* : \hat{S}\tau^*$.*

3. *Constraint generation is complete: Suppose that $\Gamma^* \vdash u \leadsto e^* : \tau^*\ [\phi]$. Let S be a substitution such that $\hat{S}\Gamma^*$ contains no type variables. If $\hat{S}\Gamma^* \vdash u \leadsto e : \tau$, then there exists substitutions S_1 and S_2 such that $S_2 = S_1 \circ S$, $\mathrm{TV}(\phi) \subseteq \mathrm{dom}(S_2)$, $S_2 \models \phi$, and $e = \hat{S}_2 e^*$ and $\tau = \hat{S}_2 \tau^*$.*

It is worthwhile to contemplate the meaning of this theorem before considering its proof. The first statement asserts that the process of constraint generation always terminates with a term scheme, type scheme, and constraint. This is easily proved by induction on the structure of u. The second statement asserts that *any* solution of the constraints determines a valid reconstruction of u, relative to the instance of the context. This is proved by rule induction on the constraint generation relation. The third statement is perhaps the hardest to understand. As a first cut, let us assume that Γ^* is empty and S is the identity. The completeness property says that *every* reconstruction of u is an instance of its schematic reconstruction by a substitution solving the constraints associated with u. Completeness is proved by induction on the elaboration relation. To complete the proof it is necessary to consider the generalization to non-empty contexts Γ^*, in the form stated.

4 Solving Constraints

solve

Constraint solving (also known as *unification*) is performed by a process similar to Guassian elimination. We accumulate a solution, S, by incrementally simplifying the constraint, gradually accumulating bindings for the variables. This process is naturally expressed by a transition system with states of the form (S, ϕ), where S is an idempotent substitution and ϕ is a constraint such that $\mathrm{dom}(S) \cap \mathrm{TV}(\phi) = \emptyset$. The restriction to idempotent substitutions means that we can regard S as a set of ordered pairs (α, τ) where $\mathrm{TV}(\tau) \cap \mathrm{dom}(S) = \emptyset$. The requirement that the type variables of ϕ be disjoint the domain of S amounts to the requirement that solved variables not occur in the remaining set of constraints.

A *state* (S, ϕ) consists of a partial solution, S, of a constraint, together with a remaining constraint that is yet to be solved to form a complete solution to the original constraint. The *initial state* is (I, ϕ), with the entire constraint remaining to be solved, and no partial solution having yet been attained for any of the type variables in ϕ. A *final state* has of the form $(S, \top)$, consisting of a complete solution, with no further constraints to be solved.

To simplify the presentation of the transition system, we impose the following structural equivalences on constraints:

$$\begin{array}{rcl}
\top \wedge \phi & \equiv & \phi \\
\phi \wedge \top & \equiv & \phi \\
\phi_1 \wedge \phi_2 & \equiv & \phi_2 \wedge \phi_1 \\
\phi_1 \wedge (\phi_2 \wedge \phi_3) & \equiv & (\phi_1 \wedge \phi_2) \wedge \phi_3 \\
\tau_1^* {=} \tau_2^* & \equiv & \tau_2^* {=} \tau_1^*
\end{array}$$

Informally, this means that we may regard a constraint as a multiset (set with repetitions) of unordered pairs of type schemes. This allows us to avoid a bunch of bookkeeping that would distract from the presentation.

The rules of the transition system are as follows.

First, equations between identical base types and identical type variables may be dropped.

$$\begin{array}{rcl}(S, \texttt{int}{=}\texttt{int} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \texttt{bool}{=}\texttt{bool} \wedge \phi) & \mapsto & (S, \phi) \\ (S, \alpha{=}\alpha \wedge \phi) & \mapsto & (S, \phi)\end{array}$$

Second, an equation between function types reduces to two equations between their respective domains and ranges.

$$(S, \tau_1^* {\rightarrow} \tau_2^* {=} \sigma_1^* {\rightarrow} \sigma_2^* \wedge \phi) \mapsto (S, \tau_1^* {=} \sigma_1^* \wedge \tau_2^* {=} \sigma_2^* \wedge \phi)$$

Third, an equation of the form $\alpha{=}\tau^*$ partially determines the solution for α, provided that α does not occur in τ^*. (This is the so-called *occurs check*.)

$$(S, \alpha{=}\tau^* \wedge \phi) \mapsto ([\alpha{=}\tau^*] \circ S, [\tau^*/\alpha]\phi)$$

provided that $\alpha \notin \mathrm{TV}(\tau^*)$. The reason for this restriction is that an equation such as $\alpha{=}\alpha{\rightarrow}\alpha$ is unsolvable, since any solution for α would have to be an infinite type expression, which is impossible.

Notice that we replace all occurrences of α in the remaining constraints ϕ by τ^* so that α no longer occurs in the constraints after this step is taken. Furthermore, we compose the singleton substitution with S to ensure that occurrences of α in the range of S are instantiated by τ^*. Both of these steps ensure that the invariants governing the state are maintained.

An *irreducible state* (S, ϕ) is one for which there is no state (S', ϕ') such that $(S, \phi) \mapsto (S', \phi')$. Clearly every final state is irreducible, but not all irreducible states are final. For example, the state $(S, \texttt{int}{=}\texttt{bool})$ is irreducible. Irreducible, non-final states are called *stuck* states. The solution process has stopped, but no solution has been found. An examination of the transition rules reveals that stuck states must consist entirely of equations of one of the following forms:

1. $\texttt{int}{=}\texttt{bool}$
2. $\tau_1^* {\rightarrow} \tau_2^* {=} \tau^*$, where τ^* is neither a function type nor a type variable.
3. $\alpha{=}\tau^*$, where $\alpha \in \mathrm{TV}(\tau^*)$.

It is obviously decidable whether or not a state is stuck. It is worthwhile to check that if a constraint ϕ contains an equation not of these three forms, then progress can be made.

An important property of the transition system is that there can be no infinite transition sequences $(S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots$. To prove this, we define a "size measure" for states that is decreased by each transition. Examining the transitions in turn, we can make the following observations:

1. The first set of transitions reduces the size of the constraint (measured as the sum of the sizes of the type schemes occurring in it).

2. The second transition also reduces the size of the constraint, since it eliminates two occurrences of the $\rightarrow$ symbol.

3. The third transition, however, can *increase* the size of the constraint because the substitution may replicate τ^*, thereby blowing up its size. However, the number of variables occurring in the constraint definitely decreases, because all occurrences of α (of which there is at least one, prior to the transition) are eliminated.

This suggests that we assign the pair of numbers (n, s) to the state (S, ϕ), where s is the sum of the sizes of the type schemes occurring in ϕ (not S!) and n is the number of type variables occurring in ϕ. If we order these pairs lexicographically by defining $(n, s) < (n', s')$ to hold iff $n < n'$ or $n = n'$ and $s < s'$,[1] then it is easy to see that each transition reduces the size measure in the sense of this ordering. In the first case n may decrease, or n may remain the same, but s decreases. In the second, n remains the same, but s decreases. In the third, s may increase dramatically, but nevertheless n decreases. Thus we have the following theorem.

Theorem 5 (Termination)
There is no infinite sequence of states $(S_1, \phi_1) \mapsto (S_2, \phi_2) \mapsto \cdots$.

The next observation concerns the relationship between successive states. A *complete solution* for a state (S, ϕ) consists of a substitution T such that $T = U \circ S$ for some U and $T \models \phi$. In other words, T is an extension of the partial solution S to a solution for ϕ. It is easy to check that stuck states have no complete solution, and that for a final state the partial solution contained within it is a complete solution.

The following technical lemma will prove useful.

Lemma 6
$S \models [\tau^*/\alpha]\phi$ *iff* $S \circ [\alpha{=}\tau^*] \models \phi$.

[1] This is the "dictionary ordering" for two-letter words, where the "letters" are natural numbers.

Proof: By induction on the structure of ϕ. ■

The key property of the transition system is that it *preserves* and *reflects* complete solutions of states.

Theorem 7
Suppose that $(S, \phi) \mapsto (S', \phi')$. Then T is a complete solution for (S, ϕ) iff T is a complete solution for (S', ϕ').

Proof: We consider each of the transitions in turn. The case of transitions that drop identical base types or identical type variables are trivial — the presence or absence of such constraints does not change the complete solution. The transition for an equation between function types is similarly straightforward; a complete solution to the left-hand state is clearly a complete solution to the right-hand state, and vice versa.

Finally, let us consider the third transition, for which $\phi = \alpha{=}\tau^* \wedge \psi$, $\phi' = [\tau^*/\alpha]\psi$, and $S' = [\alpha{=}\tau^*] \circ S$.

Suppose that $T = U' \circ S'$ and that $T \models \phi'$; we are to show that there exists U such that $T = U \circ S$ and $T \models \phi$. Taking $U = U' \circ [\alpha{=}\tau^*]$, we have that

$$\begin{array}{rcll} T & = & U' \circ S' & \text{by asssumption on } T \\ & = & U' \circ [\alpha{=}\tau^*] \circ S & \text{by assumption on } S' \\ & = & U \circ S & \text{by definition of } U \end{array}$$

Now we must show that $T \models \alpha{=}\tau^* \wedge \psi$. First, observe that by the invariants on the state, $\hat{S}(\tau^*) = \tau^*$ and $S(\alpha) = \alpha$. Next, note that

$$\begin{array}{rcll} T(\alpha) & = & \hat{U}(S(\alpha)) & \text{since } T = U \circ S \\ & = & \hat{U}(\alpha) & \text{by state invariant} \\ & = & \hat{U}'(\tau^*) & \text{by definition of } U \end{array}$$

and that

$$\begin{array}{rcll} \hat{T}(\tau^*) & = & \hat{U}(\hat{S}(\tau^*)) & \text{since } T = U \circ S \\ & = & \hat{U}(\tau^*) & \text{by state invariants} \\ & = & \hat{U}'([\tau^*/\alpha]\tau^*) & \text{by definition of } U \\ & = & \hat{U}'(\tau^*) & \text{by the occurs check, } \alpha \notin \mathrm{TV}(\tau^*) \end{array}$$

Thus $T \models \alpha{=}\tau^*$, and hence $T = T \circ [\alpha{=}\tau^*]$. But since $T \models [\tau^*/\alpha]\psi$, it follows that $T \models \psi$.

Conversely, suppose that $T = U \circ S$ and $T \models \phi$. We are to show that there exists a sustitution U' such that $T = U' \circ S'$ and $T \models \phi'$. Since

$T \models \phi$, $T(\alpha) = \hat{T}(\tau^*)$ and $T \models \psi$. Consequently, $T = T \circ [\alpha{=}\tau^*]$, and so $T \models [\tau^*/\alpha]\psi$. It remains to find U' such that $T = U' \circ S'$. First, observe that

$$\begin{array}{rcll} U(\alpha) & = & \hat{U}(S(\alpha)) & \text{by the state invariants} \\ & = & T(\alpha) & \text{since } T = U \circ S \\ & = & \hat{T}(\tau^*) & \text{since } T \models \alpha{=}\tau^* \\ & = & \hat{U}(\hat{S}(\tau^*)) & \text{since } T = U \circ S \\ & = & \hat{U}(\tau^*). & \text{by the state invariants} \end{array}$$

Now define $U'(\alpha) = \alpha$ and $U'(\beta) = U(\beta)$ (for $\beta \neq \alpha$). Note that $U' \circ [\alpha{=}\tau^*] = U$. For if $\beta \neq \alpha$, then

$$\begin{array}{rcll} (U' \circ [\alpha{=}\tau^*])(\beta) & = & U'(\beta) & \text{since } \beta \neq \alpha \\ & = & U(\beta) & \text{by definition of } U' \end{array}$$

and

$$\begin{array}{rcll} (U' \circ [\alpha{=}\tau^*])(\alpha) & = & \hat{U}'(\tau^*) & \text{by composition of substitutions} \\ & = & \hat{U}(\tau^*) & \text{since } \alpha \notin \mathrm{TV}(\tau^*) \\ & = & U(\alpha) & \text{since } U(\alpha) = \hat{U}(\tau^*) \end{array}$$

Therefore, $U' \circ S' = U \circ [\alpha{=}\tau^*] \circ S = U \circ S = T$. ■

An important corollary of Theorems 7 and 5 states that the transition system computes *principal solutions* of constraints.

Corollary 8
Suppose that $(I, \phi) \mapsto^* (S, \top)$. The substitution T solves ϕ iff $T = U \circ S$ for some substitution U.

The substitution S is called the *principal solution* (or *most general unifier*) for ϕ.
Proof: It follows from Theorem 7 by induction on the length of the transition sequence from the inital to the final state that T is a complete solution to (I, ϕ) iff T is a complete solution to $(S, \top)$. Hence $T \models \phi$ iff $T = U \circ S$ for some U. In other words, all and only the solutions to ϕ arise as specializations of S. ■

Combining the previous results we obtain the following theorem.

Theorem 9 (Unification Algorithm)
It is decidable whether or not a constraint ϕ has a solution. If ϕ is solvable, then we may find the principal solution for it.

Proof: Find an irreducible state (S, ϕ') such that $(I, \phi) \mapsto^* (S, \phi')$. If (S, ϕ) is final, then S is the principal solution for ϕ; if it is stuck, then ϕ has no solution. ∎

Combining the results of this section with those of the previous section, we obtain the following result. Given a closed context Γ and an untyped term u, we can decide whether or not u is reconstructible to a well-typed term, and, if so, we can recover all possible reconstructions of u as instances of a term and type scheme that we may obtain by a one-pass analysis of u. To decide whether u is reconstructible (relative to Γ),

1. Find e^*, τ^*, and ϕ such that $\Gamma \vdash u \rightsquigarrow e^* : \tau^* \ [\phi]$.

2. Find the principal solution S_0 (if any) of ϕ.

If ϕ has no solution, then u is not reconstructible (relative to Γ). If ϕ has a principal solution S_0, then all and only the reconstructions of u are instances of $\hat{S}_0(e^*)$ and $\hat{S}_0(\tau^*)$.