So far we have seen λ-calculus (and several extensions) without types, and we have seen simply-typed λ-calculus as well as a few extra features. These languages appeared to be almost exactly the same except for the presence of types in the latter but not the former. This similarity raises an important question: can we take a program from untyped λ-calculus and, based on how it is used, determine what type it must have in STLC (if it has any type at all)? The process by which we try to do this is called *type inference*.

1 Type Inference

Languages like Haskell and OCaml perform this sort of operation. For instance, in the expression $(f\ 3)$, the language knows that f must be a function (because it is applied to something, not because of its name). More than that, it must be a function that takes an integer as input. This usage tells us nothing about the output type, though, so all we can say right now is that f has type $\mathsf{int} \to \alpha$ for some type variable α.

If there are other occurrences of f, those will provide different constraints, and these constraints must all have a common solution for the program to have a type. For instance, a separate occurrence of the form $\mathsf{if}\ (f\ x)\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2$ would create the constraint that $f : \alpha_x \to \mathsf{bool}$ where α_x is the type of x. This constraint does not inherently conflict win the one above; if $\alpha_x = \mathsf{int}$, then giving f type $\mathsf{int} \to \mathsf{bool}$ is consistent with both uses. If, however, x has some other type, say $\mathsf{bool} \times \mathsf{bool}$, the two constraints would be inconsistent and the program would not have a valid type.

For simple type systems like the ones we have seen, if a program is well-typed, then the type can be inferred.[1] For example, consider the following program.

$$
\begin{array}{l}
\mathsf{let}\ \mathit{sqr} = \lambda z.\ z * z \\
\mathsf{in}\ \lambda f.\ \lambda x.\ \lambda y. \\
\quad \mathsf{if}\ (f\ x\ y) \\
\quad \mathsf{then}\ (\mathit{sqr}\ x) \\
\quad \mathsf{else}\ y
\end{array}
$$

The body of *sqr* applies the multiplication operator to z, so we know that $z : \mathsf{int}$, meaning $\lambda z.\ z * z : \mathsf{int} \to \mathsf{int}$ and thus $\mathit{sqr} : \mathsf{int} \to \mathsf{int}$. We also know that the type of f must be $\tau_1 \to \tau_2 \to \mathsf{bool}$ for some τ_1 and τ_2 since it is applied to two arguments and its return value is used as a conditional test. Since *sqr* is applied the variable x in the then branch, we know that $x : \mathsf{int}$, and since f is applied to x, we therefore know that τ_1 must be int, so f must have type $\mathsf{int} \to \tau_2 \to \mathsf{bool}$. Next, because the two branches of an if statement must produce the same type and $(\mathit{sqr}\ x) : \mathsf{int}$, we know that y must also have type int. Since the second argument to f is y, that means τ_2 must be int as well. Putting this all together, the type of the entire program must be $(\mathsf{int} \to \mathsf{int} \to \mathsf{bool}) \to \mathsf{int} \to \mathsf{int} \to \mathsf{int}$.

That intuition makes sense, but can we form that into a precise algorithm?

2 Unification

Type inference and pattern matching in many languages, as well as the example above, are instances of a general mechanism called *unification*. Briefly, unification is the process of taking two terms and finding a variable substitution that will take the two input terms to the same result. Pattern matching is generally done

[1] This stops being true if your type system gets complicated and powerful enough. In fact, it is possible to write a type system where type inference is an undecidable problem.

by unifying language expressions, while type inference is done by unifying type expressions. There are other applications of unification; for example, the programming language Prolog is based on it.

The essential task of unification is to find a substitution γ that unifies two terms. That is, given s and t, we would like to find some γ such that $\gamma(s) = \gamma(t)$. For example, say f and g are functions that construct terms (like application in λ-calculus or $\rightarrow$ as a type constructor), not terms that can be substituted, while x, y, z, and w can all be substituted. Given the terms

$$s = f(x, g(y)) \qquad\qquad t = f(g(z), w)$$

the substitution

$$\gamma(\cdot) = \cdot\,[x \mapsto g(z),\, w \mapsto g(y)]$$

serves as a unifier since

$$\begin{aligned}\gamma(s) &= f(x, g(y))[x \mapsto g(z),\, w \mapsto g(y)] \\ &= f(g(z), g(y)) \\ &= f(g(z), w)[x \mapsto g(z),\, w \mapsto g(y)] \\ &= \gamma(t).\end{aligned}$$

As a notation, we will write $\gamma = [x \mapsto s, y \mapsto t, \cdots]$ to indicate the *parallel* substitution, not a sequential one. In particular, given a substitution like $x[x \mapsto g(y),\, y \mapsto z]$ would produce $g(y)$, which is different then the sequential behavior of $g(z)$.

Unifiers do not necessarily exist. For example, the terms x and $f(x)$ cannot be unified; there is no substitution that can make these terms syntactically equal.

Most General Unifiers. Even when unifiers exist, they are not necessarily unique. For example, the substitution

$$\gamma' = [x \mapsto g(f(a)),\, y \mapsto f(b),\, z \mapsto f(a),\, w \mapsto g(f(b))]$$

is also a unifier for the terms s and t above. However, if a unifier does exist, there is always a *most general unifier* (MGU) that is unique up to renaming. A substitution γ is a most general unifier for s and t if

- γ is a unifier for s and t, and
- any other unifier γ' for s and t is a *refinement* of γ. That is, we can get γ' from γ by doing more substitutions.

For example, the γ above is an MGU for $f(x, g(y))$ and $f(g(z), w)$. We can get γ' by composing γ with

$$\delta = [y \mapsto f(b),\, z \mapsto f(a)].$$

We can apply the substitution directly, noting that

$$\begin{aligned}\gamma\,;\,\delta &= [x \mapsto \delta(g(z)),\, w \mapsto \delta(g(y))] \cup \delta \\ &= [x \mapsto g(f(a)),\, w \mapsto g(f(b))] \cup [y \mapsto f(b),\, z \mapsto f(a)] \\ &= \delta.\end{aligned}$$

2.1 Unification Algorithm

The unification algorithm is known as Robinson's algorithm (1965). We need a unification not just for pairs of terms, but more generally, for sets of pairs of terms. We say that γ is a unifier for the set $\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\}$ if $\gamma(s_i) = \gamma(t_i)$ for all $1 \leq i \leq n$. Note that these are *unordered* equality constraints, so $s \doteq t$ and $t \doteq s$ are considered the same.

The unification algorithm is given by the following recursive function Unif that takes a set of equality constraints and produces their MGU if one exists. If E is a set of equality constraints, then $E[x \mapsto t]$ denotes the result of apply the substitution $[x \mapsto t]$ to all terms in E.

$$\begin{aligned}
\mathsf{Unif}(\varnothing) &\triangleq I \quad \text{(the identity substitution)} \\
\mathsf{Unif}(\{x \doteq x\} \cup E) &\triangleq \mathsf{Unif}(E) \\
\mathsf{Unif}(\{f(s_1, \ldots, s_n) \doteq f(t_1, \ldots, t_n)\} \cup E) &\triangleq \mathsf{Unif}(\{s_1 \doteq t_1, \ldots, s_n \doteq t_n\} \cup E) \\
\mathsf{Unif}(\{x \doteq t\} \cup E) &\triangleq [x \mapsto t] \mathbin{;} \mathsf{Unif}(E[x \mapsto t]) \quad \text{if } x \notin \mathrm{FV}(t)
\end{aligned}$$

Let us explain these four rules.

1. The first rule says that if there is nothing left to unify, so the identity substitution—which is the most general substitution that can exists—is sufficient.
2. The second rule says that the constraint $x \doteq x$ can be simply ignored, as the terms are already the same, so nothing needs to change.
3. The third rule handles unification of compound forms. For example, if f is the type constructor $\to$, then this rule says the constraint $(s_1 \to s_2) \doteq (t_1 \to t_2)$ can be decomposed into $s_1 \doteq t_1$ and $s_2 \doteq t_2$.

   Note here that both sides of the constraint must have the same f and the same n. A constraint $f(s_1, \ldots, s_n) \doteq g(t_1, \ldots, t_m)$ where $f \neq g$ or $n \neq m$ cannot be unified.
4. In the fourth rule, we actually create part of the unifier substitution. Note the sequential composition operator, so this is the substitution $[x \mapsto t]$ *sequentially composed* with the result of $\mathsf{Unif}(E[x \mapsto t])$.

   Additionally, because we have already accounted for x, the recursive call can replace every occurrence of x with t in E. This is important because, not only does it remove x, which is already dealt with in the final substitution, but it also means that if t is a compound term, say, $f(s_1, \ldots, s_n)$, then a further constraint that $x \doteq f(t_1, \ldots, t_n)$ will result in unification of the subterms.

   Also note the requirement that $x \notin \mathrm{FV}(t)$. This is critical. Indeed, if $x \in \mathrm{FV}(t)$, then either $t = x$, in which case the second rule applies, or it is impossible to unify x and t, so Unif should fail.

3 Type Inference and Unification

We now show how to do type inference using unification on type expressions. This technique gives the *most general type* (MGT) of any typable term. That is, any other type of this term is a substituted instance of its most general type.

For this example, we will use STLC, but without any type annotations in the syntax, as the point is to infer them. Note that the syntax *with* type annotations is known as "Church style" syntax, while the syntax *without* type annotations is referred to as "Curry style." That means the syntax we are using is as follows.

$$\begin{aligned}
\tau &::= \mathsf{unit} \mid \tau_1 \to \tau_2 \\
e &::= () \mid x \mid \lambda x.\, e \mid e_1\ e_2
\end{aligned}$$

The typing rules are

$$\frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

For the language of types, the third unification rule translates to:

$$\mathsf{Unif}(\{s_1 \to s_2 \doteq t_1 \to t_2\} \cup E) \triangleq \mathsf{Unif}(\{s_1 \doteq t_1, s_2 \doteq t_2\} \cup E)$$

A problem here is that any type derivation starts with assumptions about the types of variables in the form of the type context Γ. But without a type environment or any annotations as in Church style syntax, we do not know what these are. However, we can observe that the form of the subterms impose constraints on the types. We can write down these constraints and then try to solve them.

Suppose we want to infer the type of a λ-term e. Without loss of generality, assume e has no variables bound more than once and no variable with a binding occurrence λx that also occurs free.[2]

Now let $e_1, \ldots, e_n$ be an enumeration of all subterms of e. First create a unique type variable α_i to each e_i for $1 \leq i \leq n$, and also a different unique type variable β_x for each variable x that occurs in e. Then take the following constraints.

Syntactic form	Constraint
$e_i = ()$	$\alpha_i \doteq \mathsf{unit}$
$e_i = x$	$\alpha_i \doteq \beta_x$
$e_i = \lambda x.\, e_j$	$\alpha_i \doteq \beta_x \to \alpha_j$
$e_i = e_j\ e_k$	$\alpha_j \doteq \alpha_k \to \alpha_i$

This provides a list of constraints imposed by the typing rules. We can now run Robinson's unification algorithm on these constraints to acquire a substitution γ, and then the type of the whole expression is just $\gamma(\alpha_e)$ where α_e is the type variable associated with the full expression e to get the MGT of e.

If there are more complicated types, we would have more rules for those. For instance, including integer arithmetic and boolean if statements would result in the following constraints.

Syntactic form	Constraint
$e_i = n$	$\alpha_i \doteq \mathsf{int}$
$e_i = e_j \otimes e_k$	$\alpha_i \doteq \mathsf{int}, \alpha_j \doteq \mathsf{int}, \alpha_k \doteq \mathsf{int}$
$e_i = \mathsf{true}$ or $e_i = \mathsf{false}$	$\alpha_i \doteq \mathsf{bool}$
$e_i = \mathsf{if}\ e_j\ \mathsf{then}\ e_k\ \mathsf{else}\ e_l$	$\alpha_j \doteq \mathsf{bool}, \alpha_i \doteq \alpha_k, \alpha_i \doteq \alpha_l$

3.1 An Example

To see how this works, let us infer the type of the following term:

$$e = \lambda x.\, \lambda y.\, \lambda z.\, \mathsf{if}\ x\ \mathsf{then}\ (y\ z)\ \mathsf{else}\ z$$

First we break the term into all of its subterms and generate the associated constraints:

$e_1 = \lambda x.\, \lambda y.\, \lambda z.\, \mathsf{if}\ x\ \mathsf{then}\ (y\ z)\ \mathsf{else}\ z$	$\alpha_1 \doteq \beta_x \to \alpha_2$
$e_2 = \lambda y.\, \lambda z.\, \mathsf{if}\ x\ \mathsf{then}\ (y\ z)\ \mathsf{else}\ z$	$\alpha_2 \doteq \beta_y \to \alpha_3$
$e_3 = \lambda z.\, \mathsf{if}\ x\ \mathsf{then}\ (y\ z)\ \mathsf{else}\ z$	$\alpha_3 \doteq \beta_z \to \alpha_4$
$e_4 = \mathsf{if}\ x\ \mathsf{then}\ (y\ z)\ \mathsf{else}\ z$	$\alpha_5 \doteq \mathsf{bool},\ \alpha_4 \doteq \alpha_6,\ \alpha_4 \doteq \alpha_9$
$e_5 = x$	$\alpha_5 \doteq \beta_x$
$e_6 = y\ z$	$\alpha_7 \doteq \alpha_8 \to \alpha_6$
$e_7 = y$	$\alpha_7 \doteq \beta_y$
$e_8 = z$	$\alpha_8 \doteq \beta_z$
$e_9 = z$	$\alpha_9 \doteq \beta_z$

[2]This assumption is known as the *Barendregt variable convention* after Henk Barendregt and can be achieved by a simple α-renaming.

Solving these constraints with Robinson's algorithm yields:

$$
\begin{aligned}
\alpha_1 &\mapsto \mathsf{bool} \to (\alpha_8 \to \alpha_8) \to \alpha_8 \to \alpha_8 \\
\alpha_2 &\mapsto (\alpha_8 \to \alpha_8) \to \alpha_8 \to \alpha_8 \\
\alpha_3 &\mapsto \alpha_8 \to \alpha_8 \\
\alpha_4 &\mapsto \alpha_8 \\
\alpha_5 &\mapsto \mathsf{bool} \\
\alpha_6 &\mapsto \alpha_8 \\
\alpha_7 &\mapsto \alpha_8 \to \alpha_8 \\
\alpha_9 &\mapsto \alpha_8 \\
\beta_x &\mapsto \mathsf{bool} \\
\beta_y &\mapsto \alpha_8 \to \alpha_8 \\
\beta_z &\mapsto \alpha_8
\end{aligned}
$$

That means the MGT of the expression is $\tau = \mathsf{bool} \to (\alpha \to \alpha) \to \alpha \to \alpha$ for a type variable α. In other words, no matter what type τ' we put in for α, the term e can be well-typed at type $\tau[\alpha \mapsto \tau']$.