Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Notes on Typed Lambda Calculi

Spring 2017

A *type* is a collection of computational entities that share some common property. For example, the type **int** represents all expressions that evaluate to an integer, and the type **int** $\to$ **int** represents all functions from integers to integers. The Pascal subrange type `[1..100]` represents all integers between 1 and 100.

Types can be thought of as describing computations succinctly and approximately: types are a *static* approximation to the run-time behaviors of terms and programs. Type systems are a lightweight formal method for reasoning about behavior of a program. Uses of type systems include: naming and organizing useful concepts; providing information (to the compiler or programmer) about data manipulated by a program; and ensuring that the run-time behavior of programs meet certain criteria.

In this lecture, we'll consider a type system for the lambda calculus that ensures that values are used correctly; for example, that a program never tries to add an integer to a function. The resulting language (lambda calculus plus the type system) is called the *simply-typed lambda calculus*.

1 Simply-typed lambda calculus

The syntax of the simply-typed lambda calculus is similar to that of untyped lambda calculus, with the exception of abstractions. Since abstractions define functions that take an argument, in the simply-typed lambda calculus, we explicitly state what the type of the argument is. That is, in an abstraction $\lambda x : \tau.\, e$, the τ is the expected type of the argument.

The syntax of the simply-typed lambda calculus is as follows. We will include integer literals n, addition $e_1 + e_2$, and the *unit value* $()$. The unit value is the only value of type **unit**.

expressions	$e ::= x \mid \lambda x : \tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid ()$
values	$v ::= \lambda x : \tau.\, e \mid n \mid ()$
types	$\tau ::= \textbf{int} \mid \textbf{unit} \mid \tau_1 \to \tau_2$

The operational semantics of the simply-typed lambda calculus are the same as the untyped lambda calculus. For completeness, we present the CBV small step operational semantics here.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E + e \mid v + E$$

$$\textsc{Context}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}$$

$$\beta\text{-}\textsc{reduction}\ \frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\textsc{Add}\ \frac{}{n_1 + n_2 \longrightarrow n}\ n = n_1 + n_2$$

1.1 The typing relation

The presence of types does not alter the evaluation of an expression at all. So what use are types?

We will use types to restrict what expressions we will evaluate. Specifically, the type system for the simply-typed lambda calculus will ensure that any *well-typed* program will not get *stuck*. A term e is stuck if e is not a value and there is no term e' such that $e \longrightarrow e'$. For example, the expression $42 + \lambda x : \textbf{int}.\, x$ is stuck: it attempts to add an integer and a function; it is not a value, and there is no operational rule that allows us to reduce this expression. Another stuck expression is $()\ 47$, which attempts to apply the unit value to an integer.

We introduce a relation (or *judgment*) over *typing contexts* (or *type environments*) Γ, expressions e, and types τ. The judgment

$$\Gamma \vdash e : \tau$$

is read as "e has type τ in context Γ".

A typing context is a sequence of variables and their types. In the typing judgment $\Gamma \vdash e : \tau$, we will ensure that if x is a free variable of e, then Γ associates x with a type. We can view a typing context as a partial function from variables to types. We will write $\Gamma, x : \tau$ or $\Gamma[x \mapsto \tau]$ to indicate the typing context that extends Γ by associating variable x with with type τ. The empty context is sometimes written $\varnothing$, or often just not written at all. For example, we write $\vdash e : \tau$ to mean that the closed term e has type τ under the empty context.

Given a typing environment Γ and expression e, if there is some τ such that $\Gamma \vdash e : \tau$, we say that e *is well-typed under context* Γ; if Γ is the empty context, we say e *is well-typed.*

We define the judgment $\Gamma \vdash e : \tau$ inductively.

$$\text{T-Int}\ \frac{}{\Gamma \vdash n : \mathbf{int}} \qquad \text{T-Add}\ \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \qquad \text{T-Unit}\ \frac{}{\Gamma \vdash () : \mathbf{unit}}$$

$$\text{T-Var}\ \frac{}{\Gamma \vdash x : \tau}\ \Gamma(x) = \tau \qquad \text{T-Abs}\ \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.\, e : \tau \to \tau'} \qquad \text{T-App}\ \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

An integer n always has type **int**. Expression $e_1 + e_2$ has type **int** if both e_1 and e_2 have type **int**. The unit value () always has type **unit**.

Variable x has whatever type the context associates with x. Note that Γ must contain an associating for x in order to the judgment $\Gamma \vdash x : \tau$ to hold, that is, $x \in \mathrm{dom}(\Gamma)$. The abstraction $\lambda x : \tau.\, e$ has the function type $\tau \to \tau'$ if the function body e has type τ' under the assumption that x has type τ. Finally, an application $e_1\ e_2$ has type τ' provided that e_1 is a function of type $\tau \to \tau'$, and e_2 is an argument of the expected type, i.e., of type τ.

To type check an expression e, we attempt to construct a derivation of the judgment $\vdash e : \tau$, for some type τ. For example, consider the program $(\lambda x : \mathbf{int}.\, x + 40)\ 2$. The following is a proof that $(\lambda x : \mathbf{int}.\, x + 40)\ 2$ is well-typed.

$$\text{T-App}\ \frac{\text{T-Abs}\ \dfrac{\text{T-Add}\ \dfrac{\text{T-Var}\ \dfrac{}{x : \mathbf{int} \vdash x : \mathbf{int}} \quad \text{T-Int}\ \dfrac{}{x : \mathbf{int} \vdash 40 : \mathbf{int}}}{x : \mathbf{int} \vdash x + 40 : \mathbf{int}}}{\vdash \lambda x : \mathbf{int}.\, x + 40 : \mathbf{int} \to \mathbf{int}} \quad \text{T-Int}\ \dfrac{}{\vdash 2 : \mathbf{int}}}{\vdash (\lambda x : \mathbf{int}.\, x + 40)\ 2 : \mathbf{int}}$$

1.2 Type soundness

We mentioned above that the type system ensures that any well-typed program does not get stuck. We can state this property formally.

Theorem (Type soundness)**.** *If $\vdash e : \tau$ and $e \longrightarrow^* e'$ then either e' is a value, or there exists e'' such that $e' \longrightarrow e''$.*

We will prove this theorem using two lemmas: *preservation* and *progress.* Intuitively, preservation says that if an expression e is well-typed, and e can take a step to e', then e' is well-typed. That is, evaluation preserves well-typedness. Progress says that if an expression e is well-typed, then either e is a value, or there is an e' such that e can take a step to e'. That is, well-typedness means that the expression cannot get stuck. Together, these two lemmas suffice to prove type soundness.

1.2.1 Preservation

Lemma (Preservation)**.** *If $\vdash e : \tau$ and $e \longrightarrow e'$ then $\vdash e' : \tau$.*

Proof. We proceed by induction on $e \longrightarrow e'$. That is, we will prove for all e and e' such that $e \longrightarrow e'$, that $P(e \longrightarrow e')$ holds, where

$$P(e \longrightarrow e') = \forall \tau.\ \text{if } \vdash e : \tau \text{ then } \vdash e' : \tau.$$

Consider each of the inference rules for the small step relation.

- Add

  Assume $\vdash e : \tau$.

  Here $e \equiv n_1 + n_2$, and $e' = n$ where $n = n_1 + n_2$, and $\tau = \textbf{int}$. By the typing rule T-Int, we have $\vdash e' : \textbf{int}$ as required.

- β-reduction

  Assume $\vdash e : \tau$.

  Here, $e \equiv (\lambda x : \tau'.\, e_1)\ v$ and $e' \equiv e_1\{v/x\}$. Since e is well-typed, we have derivations showing $\vdash \lambda x : \tau'.\, e_1 : \tau' \to \tau$ and $\vdash v : \tau'$. There is only one typing rule for abstractions, T-Abs, from which we know $x : \tau \vdash e_1 : \tau$. By the substitution lemma (see below), we have $\vdash e_1\{v/x\} : \tau$ as required.

- Context

  Assume $\vdash e : \tau$.

  Here, we have some context E such that $e = E[e_1]$ and $e' = E[e_2]$ for some e_1 and e_2 such that $e_1 \longrightarrow e_2$. The inductive hypothesis is that $P(e_1 \longrightarrow e_2)$.

  Since e is well-typed, we can show by induction on the structure of E that $\vdash e_1 : \tau_1$ for some τ_1. By the inductive hypothesis, we thus have $\vdash e_2 : \tau_1$. By the context lemma (see below) we have $\vdash E[e'] : \tau$ as required.

□

Additional lemmas we used in the proof above.

Lemma (Substitution)**.** *If $x : \tau' \vdash e : \tau$ and $\vdash v : \tau'$ then $\vdash e\{v/x\} : \tau$.*

Lemma (Context)**.** *If $\vdash E[e_0] : \tau$ and $\vdash e_0 : \tau'$ and $\vdash e_1 : \tau'$ then $\vdash E[e_1] : \tau$.*

1.2.2 Progress

Lemma (Progress)**.** *If $\vdash e : \tau$ then either e is a value or there exists an e' such that $e \longrightarrow e'$.*

Proof. We proceed by induction on the derivation of $\vdash e : \tau$. That is, we will show for all e and τ such that $\vdash e : \tau$, we have $P(\vdash e : \tau)$, where

$$P(\vdash e : \tau) = \text{either } e \text{ is a value or } \exists e' \text{ such that } e \longrightarrow e'.$$

- T-Var

  This case is impossible, since a variable is not well-typed in the empty environment.

- T-Unit, T-Int, T-Abs

  Trivial, since e must be a value.

- T-Add

  Here $e \equiv e_1 + e_2$ and $\vdash e_i : \textbf{int}$ for $i \in \{1, 2\}$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \longrightarrow e_i'$.

  If e_1 is not a value, then by Context, $e_1 + e_2 \longrightarrow e_1' + e_2$. If e_1 is a value and e_2 is not a value, then by Context, $e_1 + e_2 \longrightarrow e_1 + e_2'$. If e_1 and e_2 are values, then, it must be the case that they are both integer literals, and so, by Add, we have $e_1 + e_2 \longrightarrow n$ where n equals e_1 plus e_2.

- T-App

  Here $e \equiv e_1\ e_2$ and $\vdash e_1 : \tau' \to \tau$ and $\vdash e_2 : \tau'$. By the inductive hypothesis, for $i \in \{1, 2\}$, either e_i is a value or there is an e_i' such that $e_i \longrightarrow e_i'$.

  If e_1 is not a value, then by Context, $e_1\ e_2 \longrightarrow e_1'\ e_2$. If e_1 is a value and e_2 is not a value, then by Context, $e_1\ e_2 \longrightarrow e_1\ e_2'$. If e_1 and e_2 are values, then, it must be the case that e_1 is an abstraction $\lambda x : \tau'.\, e'$, and so, by β-reduction, we have $e_1\ e_2 \longrightarrow e'\{e_2/x\}$.

□

1.3 Expressive power of the simply-typed lambda calculus

Clearly, not all expressions in the untyped lambda calculus are well-typed. Indeed, type soundness implies that any lambda calculus program that gets stuck is not well-typed. But are there programs that do not get stuck that are not well-typed?

Unfortunately, the answer is yes.

First, since the simply-typed lambda calculus requires us to specify a type for function arguments, any given function can only take arguments of one type. Consider, for example, the identity function $\lambda x.\, x$. This function may be applied to any argument, and it will not get stuck. However, we must provide a type for the argument. If we specify $\lambda x : \mathbf{int}.\, x$, then this function can only accept integers, and the program $(\lambda x : \mathbf{int}.\, x)\ ()$ is not well-typed, even though it does not get stuck. Indeed, in the simply-typed lambda calculus, there is a different identity function for each type.

Second, we can no longer write recursive functions. Consider the nonterminating expression $\Omega = (\lambda x.\, x\ x)\ (\lambda x.\, x\ x)$. What type does it have? Let's suppose that the type of $\lambda x.\, x\ x$ is $\tau \to \tau'$. But $\lambda x.\, x\ x$ is applied to itself! So that means that the type of $\lambda x.\, x\ x$ is the argument type τ. So we have that τ must be equal to $\tau \to \tau'$. There is no such type for which this equality holds. (At least, not in this type system...)

This means that every well-typed program in the simply-typed lambda calculus terminates. More formally:

Theorem 1 (Normalization)**.** *If $\vdash e : \tau$ then there exists a value v such that $e \longrightarrow^* v$.*

This is known as *normalization* since it means that given any well-typed expression, we can reduce it to a *normal form*, which, in our case, is a value.

2 Products and sums

We have previously seen *products*, which are pairs of expressions. Products were constructed using the expression (e_1, e_2), and destructed using projection $\#1\ e$ and $\#2\ e$.

$$e ::= \cdots \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e$$
$$v ::= \cdots \mid (v_1, v_2)$$

Again, there are structural rules to determine the order of evaluation. In a CBV lambda calculus, the evaluation contexts are extended as follows.

$$E ::= \cdots \mid (E, e) \mid (v, E) \mid \#1\ E \mid \#2\ E$$

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructor interact.

$$\frac{}{\#1\ (v_1, v_2) \longrightarrow v_1} \qquad \frac{}{\#2\ (v_1, v_2) \longrightarrow v_2}$$

The type of a product expression (or a *product type*) is a pair of types, written $\tau_1 \times \tau_2$. The typing rules for the product constructors and destructors are the following.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#1\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \#2\ e : \tau_2}$$

We introduce *sums*, which are dual to products. Intuitively, a product holds two values, one of type τ_1, and one of type τ_2. By contrast, a sum holds a single value that is either of type τ_1 or of type τ_2. The type of a sum is written $\tau_1 + \tau_2$. There are two constructors for a sum, corresponding to whether we are constructing a sum with a value of τ_1 or a value of τ_2.

$$\begin{aligned} e &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 \\ v &::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v \end{aligned}$$

Again, there are structural rules to determine the order of evaluation. In a CBV lambda calculus, the evaluation contexts are extended as follows.

$$E ::= \cdots \mid \mathsf{inl}_{\tau_1+\tau_2}\ E \mid \mathsf{inr}_{\tau_1+\tau_2}\ E \mid \mathsf{case}\ E\ \mathsf{of}\ e_2 \mid e_3$$

In addition to the structural rules, there are two operational semantics rules that show how the destructors and constructors interact.

$$\frac{}{\mathsf{case}\ \mathsf{inl}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \longrightarrow e_2\ v} \qquad \frac{}{\mathsf{case}\ \mathsf{inr}_{\tau_1+\tau_2}\ v\ \mathsf{of}\ e_2 \mid e_3 \longrightarrow e_3\ v}$$

The type of a sum expression (or a *sum type*) is written $\tau_1 + \tau_2$. The typing rules for the sum constructors and destructor are the following.

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_1 + \tau_2 \quad \Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \to \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ e_1 \mid e_2 : \tau}$$

Let's see an example of a program that uses sum types.

$$\begin{aligned} &\mathsf{let}\ f : (\mathbf{int} + (\mathbf{int} \to \mathbf{int})) \to \mathbf{int} = \\ &\qquad \lambda a : \mathbf{int} + (\mathbf{int} \to \mathbf{int}).\ \mathsf{case}\ a\ \mathsf{of}\ \lambda y.\ y + 1 \mid \lambda g.\ g\ 35\ \mathsf{in} \\ &\mathsf{let}\ h : \mathbf{int} \to \mathbf{int} = \lambda x : \mathbf{int}.\ x + 7\ \mathsf{in} \\ &f\ (\mathsf{inr}_{\mathbf{int}+(\mathbf{int}\to\mathbf{int})}\ h) \end{aligned}$$

Here, the function f takes argument a, which is a sum. That is, the actual argument for a will either be a value of type **int** or a value of type **int** → **int**. We destroy the sum value with a case statement, which must be prepared to take either of the two kinds of values that the sum may contain. We end up applying f to a value of type **int** → **int** (i.e., a value injected into the right type of the sum). The entire program ends up evaluating to 42.

3 Recursion

We saw in last lecture that we could not type recursive functions or fixed-point combinators in the simply-typed lambda calculus. So instead of trying (and failing) to define a fixed-point combinator in the simply-typed lambda calculus, we add a new primitive $\mu x : \tau.\ e$ to the language. The evaluation rules for the new primitive will mimic the behavior of fixed-point combinators.

We extend the syntax with the new primitive operator. Intuitively, $\mu x : \tau.\ e$ is the fixed-point of the function $\lambda x : \tau.\, e$. Note that $\mu x : \tau.\ e$ is *not* a value, regardless of whether e is a value or not.

$$e ::= \cdots \mid \mu x : \tau.\ e$$

We extend the operational semantics for the new operator. There is a new axiom, but no new evaluation contexts.

$$\frac{}{\mu x{:}\tau.\ e \longrightarrow e\{(\mu x{:}\tau.\ e)/x\}}$$

Note that we can define the $\mathsf{letrec}\ x{:}\tau = e_1\ \mathsf{in}\ e_2$ construct in terms of this new expression.

$$\mathsf{letrec}\ x{:}\tau = e_1\ \mathsf{in}\ e_2 \triangleq \mathsf{let}\ x{:}\tau = \mu x{:}\tau.\ e_1\ \mathsf{in}\ e_2$$

We add a new typing rule for the new language construct.

$$\frac{\Gamma[x \mapsto \tau] \vdash e{:}\tau}{\Gamma \vdash \mu x{:}\tau.\ e{:}\tau}$$

Returning to our trusty factorial example, the following program implements the factorial function using the $\mu x{:}\tau.\ e$ expression.

$$FACT \triangleq \mu f{:}\mathbf{int} \to \mathbf{int}.\ \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (f\ (n-1))$$

Or using our convenient letrec notation, we could define a variable *fact* as follows.

$$\begin{array}{l}\mathsf{letrec}\ \mathit{fact}{:}\mathbf{int} \to \mathbf{int} = \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (\mathit{fact}\ (n-1)) \\ \mathsf{in}\ \ldots\end{array}$$

We can write non-terminating computations for any type: the expression $\mu x{:}\tau.\ x$ has type τ, and does not terminate.

Although the $\mu x{:}\tau.\ e$ expression is normally used to define recursive functions, it can be used to find fixed points of any type. For example, consider the following expression.

$$\begin{array}{rl}\mu x{:}(\mathbf{int} \to \mathbf{bool}) \times (\mathbf{int} \to \mathbf{bool}).\ & (\lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{true}\ \mathsf{else}\ ((\#2\ x)\ (n-1)), \\ & \lambda n{:}\mathbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ \mathsf{false}\ \mathsf{else}\ ((\#1\ x)\ (n-1)))\end{array}$$

This expression has type $(\mathbf{int} \to \mathbf{bool}) \times (\mathbf{int} \to \mathbf{bool})$—it is a pair of mutually recursive functions; the first function returns true only if its argument is even; the second returns true only if its argument is odd.

4 Denotational semantics

The denotational semantics of the simply-typed lambda calculus is much more straightforward than that of the untyped lambda calculus, because types provide helpfull structure.

We first consider the CBV simply-typed lambda calculus with products and sums:

$$\begin{array}{l}e ::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \mid () \mid (e_1, e_2) \mid \#1\ e \mid \#2\ e \mid \mathsf{inl}_{\tau_1+\tau_2}\ e \mid \mathsf{inr}_{\tau_1+\tau_2}\ e \mid \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 \\ v ::= \lambda x{:}\tau.\, e \mid n \mid () \mid (v_1, v_2) \mid \mathsf{inl}_{\tau_1+\tau_2}\ v \mid \mathsf{inr}_{\tau_1+\tau_2}\ v \\ \tau ::= \mathbf{int} \mid \mathbf{unit} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2\end{array}$$

From the operational semantics, we know that well-typed terms reduce to values (by strong normalization) of the same type as the original expression (by type soundness). Thus, a reasonable denotational semantics is to associate with every term the resulting value to which it reduces. Thus, the semantics associates with terms of type τ results of type τ. The map $[\![\tau]\!]$ describes the domain of type τ:

$$\begin{aligned}[\![\mathbf{int}]\!] &= \mathbb{Z} \\ [\![\mathbf{unit}]\!] &= \{\bullet\} \\ [\![\tau_1 \to \tau_2]\!] &= [\![\tau_1]\!] \to [\![\tau_2]\!] \\ [\![\tau_1 \times \tau_2]\!] &= [\![\tau_1]\!] \times [\![\tau_2]\!] \\ [\![\tau_1 + \tau_2]\!] &= [\![\tau_1]\!] \uplus [\![\tau_2]\!]\end{aligned}$$

where for the sake of concreteness we take $X \uplus Y = \{(0, x) \mid x \in X\} \cup \{(1, y) \mid y \in Y\}$.

Since we only care about well-typed terms, we only give a semantics to such. We will do so by assigning a meaning to *typing judgments*: we define a semantics $[\![\Gamma \vdash e : \tau]\!]$. Since the semantics is defined in terms of a typing derivation, we have the subtrees of the derivation available to define the semantics of the subterms.

To account for open terms (since subterms of even a closed terms may be open), the meaning of a term e of type τ will be a function from environments (giving a value for every variable) to values in $[\![\tau]\!]$.

An environment ρ is a map

$$\mathbf{Var} \to \bigcup \{[\![\tau]\!] \mid \tau \text{ a type}\}$$

assigning to every variable a value (of any type). Let $\mathbf{Env}_\Gamma$ be the set of environments that respect the typing context Γ: the set of all ρ such that for all $x \in Dom(\Gamma)$, $\rho(x) \in [\![\Gamma(\tau)]\!]$.

$$[\![\Gamma \vdash e : \tau]\!] : \mathbf{Env}_\Gamma \to [\![\tau]\!]$$

$$\begin{aligned}
[\![\Gamma \vdash x : \tau]\!]\rho &= \rho(x) \\
[\![\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau']\!]\rho &= \boldsymbol{\lambda} v \in [\![\tau]\!].[\![\Gamma, x{:}\tau \vdash e : \tau']\!]\rho[x \mapsto v] \\
[\![\Gamma \vdash e_1\ e_2 : \tau]\!]\rho &= ([\![\Gamma \vdash e_1 : \tau' \to \tau]\!]\rho)\ ([\![\Gamma \vdash e_2 : \tau']\!]\rho) \\
[\![\Gamma \vdash n : \mathsf{int}]\!]\rho &= n \\
[\![\Gamma \vdash e_1 + e_2 : \mathsf{int}]\!]\rho &= [\![\Gamma \vdash e_1 : \mathsf{int}]\!]\rho + [\![\Gamma \vdash e_2 : \mathsf{int}]\!]\rho \\
[\![\Gamma \vdash () : \mathsf{unit}]\!]\rho &= \bullet \\
[\![\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2]\!]\rho &= ([\![\Gamma \vdash e_1 : \tau_1]\!]\rho, [\![\Gamma \vdash e_2 : \tau_2]\!]\rho) \\
[\![\Gamma \vdash \#1\ e : \tau_1]\!]\rho &= \pi_1([\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\rho) \\
[\![\Gamma \vdash \#2\ e : \tau_2]\!]\rho &= \pi_2([\![\Gamma \vdash e : \tau_1 \times \tau_2]\!]\rho) \\
[\![\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2]\!]\rho &= (0, [\![\Gamma \vdash e : \tau_1]\!]\rho) \\
[\![\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e : \tau_1 + \tau_2]\!]\rho &= (1, [\![\Gamma \vdash e : \tau_2]\!]\rho) \\
[\![\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3 : \tau]\!]\rho &= [[\![\Gamma \vdash e_2 : \tau_1 \to \tau]\!]\rho, [\![\Gamma \vdash e_3 : \tau_2 \to \tau]\!]\rho]\ \ ([\![\Gamma \vdash e_1 : \tau_1 + \tau_2]\!]\rho)
\end{aligned}$$

where $[f, g]$ is defined by:

$$[f, g]\ (i, v) = \begin{cases} f(v) & \text{if } i = 0 \\ g(v) & \text{if } i = 1 \end{cases}$$

Thus, roughly, functions in the simply-typed lambda calculus correspond to functions over type domains, products in the simply-typed lambda calculus correspond to Cartesian products over type domains, and sums in the simply-typed lambda calculus correspond to disjoin unions over type domains. All in all, a fairly natural correspondence.

To what extent does the above denotational semantics capture the operational semantics? The following result is reasonably easy to prove, by structural induction on the reduction relation $\longrightarrow$:

Theorem 2. *If $\vdash e : \tau$ and $e \longrightarrow e'$, then $[\![\vdash e : \tau]\!] = [\![\vdash e' : \tau]\!]$.*

An easy induction then yields

Corollary 1. *If $\vdash e : \tau$ and $e \longrightarrow^* v$, then $[\![\vdash e : \tau]\!] = [\![\vdash v : \tau]\!]$.*

In other words, the denotational semantics of a term is preserved by the operational semantics. What about the other direction? If an expression and a value have the same denotation, does the expression reduce to the value? Unfortunately, that result does *not* hold in general. Consider $e = \lambda x : \mathsf{int}.\, x + 0$ and $v = \lambda x : \mathsf{int}.\, x$. It is easy to see that $[\![\vdash e : \mathsf{int} \to \mathsf{int}]\!] = [\![\vdash v : \mathsf{int} \to \mathsf{int}]\!] = \boldsymbol{\lambda} v \in \mathbb{Z}.v$, but we cannot have $e \longrightarrow^* v$ since e is already a value and does not reduce any further.

What we *can* show is that as long as type τ does not contain a function type, then the converse to Corollary 1 holds.

Theorem 3. *If $\vdash e : \tau$ for τ not containing a function type and $[\![\vdash e : \tau]\!] = [\![\vdash v : \tau]\!]$, then $e \longrightarrow^* v$.*

The proof of this theorem is a bit more involved, though, and requires the use of a new technique called *logical relations*. (See Chapter 11 of Winskell's *The Formal Semantics of Programming Languages*, for example.)

What about recursion? If we add $\mu x\!:\!\tau.\ e$ to the language, how can we adapt the denotational semantics? Just like we did for the denotational semantics of IMP, one way is to move to partial functions.

To simplify the presentation somewhat, we first restrict recursion so that in $\mu x : \tau.\ e$ expression e is actually an abstraction $\lambda y\!:\!\tau_1.\, e$. (In a call-by-value language, the only useful recursive artifacts are recursive functions.) Thus, we restrict recursive expressions to $\mu x\!:\!\tau_1 \to \tau_2.\lambda y\!:\!\tau_1.\, e$.

The domain of type $\tau_1 \to \tau_2$ is now the set $[\![\tau_1]\!] \rightharpoonup [\![\tau_2]\!]$ of partial functions from $[\![\tau_1]\!]$ to $[\![\tau_2]\!]$, and the denotation of a judgment $\Gamma \vdash e\!:\!\tau$ is now a partial function from environments to the domain $[\![\tau]\!]$:

$$[\![\Gamma \vdash e\!:\!\tau]\!] : \mathbf{Env}_\Gamma \rightharpoonup [\![\tau]\!]$$

The denotational semantics are now an adapation of the above denotational to partial functions. As we have done in the past, we freely use the graph representation of partial functions to simplify the definition of partial functions.

$$\begin{aligned}
[\![\Gamma \vdash x : \tau]\!] &= \{(\rho, \rho(x)) \mid \rho \in \mathbf{Env}_\Gamma\} \\
[\![\Gamma \vdash \lambda x\!:\!\tau.\, e\!:\!\tau \to \tau']\!] &= \{(\rho, f) \mid \rho \in \mathbf{Env}_\Gamma, f = \{(v, v') \mid (\rho[x \mapsto v], v') \in [\![\Gamma, x\!:\!\tau \vdash e\!:\!\tau']\!]\}\} \\
[\![\Gamma \vdash e_1\ e_2\!:\!\tau]\!] &= \{(\rho, v') \mid \rho \in \mathbf{Env}_\Gamma, (\rho, f) \in [\![\Gamma \vdash e_1\!:\!\tau' \to \tau]\!], (\rho, v) \in [\![\Gamma \vdash e_2\!:\!\tau']\!], (v, v') \in f\} \\
[\![\Gamma \vdash n\!:\!\mathbf{int}]\!] &= \{(\rho, n) \mid \rho \in \mathbf{Env}_\Gamma\} \\
[\![\Gamma \vdash e_1 + e_2\!:\!\mathbf{int}]\!] &= \{(\rho, n) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, n_1) \in [\![\Gamma \vdash e_1\!:\!\mathbf{int}]\!], (\rho, n_2) \in [\![\Gamma \vdash e_2\!:\!\mathbf{int}]\!], n = n_1 + n_2\} \\
[\![\Gamma \vdash ()\!:\!\mathbf{unit}]\!] &= \{(\rho, \bullet) \mid \rho \in \mathbf{Env}_\Gamma\} \\
[\![\Gamma \vdash (e_1, e_2)\!:\!\tau_1 \times \tau_2]\!] &= \{(\rho, (v_1, v_2)) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, v_1) \in [\![\Gamma \vdash e_1\!:\!\tau_1]\!], (\rho, v_2) \in [\![\Gamma \vdash e_2\!:\!\tau_2]\!]\} \\
[\![\Gamma \vdash \#1\ e\!:\!\tau_1]\!] &= \{(\rho, v_1) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, (v_1, v_2)) \in [\![\Gamma \vdash e\!:\!\tau_1 \times \tau_2]\!]\} \\
[\![\Gamma \vdash \#2\ e\!:\!\tau_2]\!] &= \{(\rho, v_2) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, (v_1, v_2)) \in [\![\Gamma \vdash e\!:\!\tau_1 \times \tau_2]\!]\} \\
[\![\Gamma \vdash \mathsf{inl}_{\tau_1+\tau_2}\ e\!:\!\tau_1 + \tau_2]\!] &= \{(\rho, (0, v)) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, v) \in [\![\Gamma \vdash e\!:\!\tau_1]\!])\} \\
[\![\Gamma \vdash \mathsf{inr}_{\tau_1+\tau_2}\ e\!:\!\tau_1 + \tau_2]\!] &= \{(\rho, (1, v)) \mid \rho \in \mathbf{Env}_\Gamma, (\rho, v) \in [\![\Gamma \vdash e\!:\!\tau_2]\!])\} \\
[\![\Gamma \vdash \mathsf{case}\ e_1\ \mathsf{of}\ e_2 \mid e_3\!:\!\tau]\!] &= \{(\rho, v') \mid\ \rho \in \mathbf{Env}_\Gamma, (\rho, (0, v)) \in [\![\Gamma \vdash e_1\!:\!\tau_1 + \tau_2]\!], \\
&\qquad\qquad (\rho, f) \in [\![\Gamma \vdash e_2\!:\!\tau_1 \to \tau]\!], (v, v') \in f\} \\
&\quad \cup \{(\rho, v') \mid\ \rho \in \mathbf{Env}_\Gamma, (\rho, (1, v)) \in [\![\Gamma \vdash e_1\!:\!\tau_1 + \tau_2]\!], \\
&\qquad\qquad (\rho, f) \in [\![\Gamma \vdash e_3\!:\!\tau_2 \to \tau]\!], (v, v') \in f\}
\end{aligned}$$

For recursive terms, we first notice that a recursive term $\mu x : \tau_1 \to \tau_2.\lambda y : \tau_1.\, e$ reduces in one step to $\lambda y\!:\!\tau_1.\, e\{\mu x\!:\!\tau_1 \to \tau_2.\ e/x\}$, which is value. Therefore, the semantics of $\mu x : \tau_1 \to \tau_2.\lambda y : \tau_1.\, e$ should be a *total* function from environments to values in $[\![\tau_1]\!] \rightharpoonup [\![\tau_2]\!]$.

This lets us derive the following equation for the semantics of $\mu x\!:\!\tau_1 \to \tau_2.\lambda y\!:\!\tau_1.\, e$:

$$[\![\Gamma \vdash (\mu x\!:\!\tau_1 \to \tau_2.\lambda y\!:\!\tau_1.\, e)\!:\!\tau_1 \to \tau_2]\!]\rho = [\![\Gamma \vdash \lambda y\!:\!\tau_1.\, e\!:\!\tau_1 \to \tau_2]\!]\rho[x \mapsto [\![\Gamma \vdash (\mu x\!:\!\tau_1 \to \tau_2.\lambda y\!:\!\tau_1.\, e)\!:\!\tau_1 \to \tau_2]\!]\rho]$$

which is actually a family of equations parameterized by ρ, of the form:

$$S_\rho = [\![\Gamma \vdash \lambda y\!:\!\tau_1.\, e\!:\!\tau_1 \to \tau_2]\!]\rho[x \mapsto S_\rho]$$

and for each such equation a solution can be found by using the techniques we saw when studying the denotational semantics of IMP, that is, taking the functional

$$G_{\Gamma,x,y,\tau_1,\tau_2,e,\rho}(f) = [\![\Gamma \vdash \lambda y\!:\!\tau_1.\, e\!:\!\tau_1 \to \tau_2]\!]\rho[x \mapsto f]$$

taking partial functions to partial functions, showing that it is continuous (it is, but it's a bear to prove directly), and therefore obtaining the solution:

$$\mathit{fix}(G_{\Gamma,x,y,\tau_1,\tau_2,e,\rho}) = \bigcup_{i \geq 0} G^i_{\Gamma,x,y,\tau_1,\tau_2,e,\rho}(\varnothing)$$

The final semantics of recursive terms is therefore:

$$[\![\Gamma \vdash (\mu x{:}\tau_1 \to \tau_2 . \lambda y{:}\tau_1 .\, e){:}\tau_1 \to \tau_2]\!]\rho = \mathit{fix}(G_{\Gamma,x,y,\tau_1,\tau_2,e,\rho})$$

As an example, work out that the semantics of the factorial function:

$$[\![\vdash (\mu f{:}\textbf{int} \to \textbf{int}.\ \lambda n{:}\textbf{int}.\ \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \times (f\ (n-1))){:}\textbf{int} \to \textbf{int}]\!]\rho$$

yields the function given by the graph

$$\{(0,1),(1,1),(2,2),(3,6),(4,24),\dots\}$$

5 Type inference

In the simply typed lambda calculus, we must explicitly state the type of function arguments: $\lambda x{:}\tau.\, e$. This explicitness makes it possible to type check functions.

$$\frac{\Gamma, x{:}\tau \vdash e{:}\tau'}{\Gamma \vdash \lambda x{:}\tau.\, e{:}\tau \to \tau'}$$

Suppose we didn't want to provide type annotations for function arguments. Consider the typing rule for functions without type annotations.

$$\frac{\Gamma, x{:}\tau \vdash e{:}\tau'}{\Gamma \vdash \lambda x.\, e{:}\tau \to \tau'}$$

The type-checking algorithm would need to guess or somehow know what type τ to put into the type context.

Can we still type check our program without these type annotations? For the simply typed lambda calculus (and many of the extensions we have considered so far), the answer is yes: we can *infer* or *reconstruct* the types of a program.

Let's consider an example to see how this type inference could work.

$$\lambda a.\, \lambda b.\, \lambda c.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

Since the variable b is used in an addition, the type of b must be **int**. The variable a must be some kind of function, since it is applied to the expression $b+1$. Since a has a function type, the type of the expression $b+1$ (i.e., **int**) must be a's argument type. Moreover, the result of the function application $(a\ (b+1))$ is used as the test of a conditional, so it better be the case that the result type of a is also **bool**. So the type of a should be **int** $\to$ **bool**. Both branches of a conditional should return values of the same type, so the type of c must be the same as the type of b, namely **int**.

We can write the expression with the reconstructed types:

$$\lambda a{:}\textbf{int} \to \textbf{bool}.\, \lambda b{:}\textbf{int}.\, \lambda c{:}\textbf{int}.\, \mathsf{if}\ a\ (b+1)\ \mathsf{then}\ b\ \mathsf{else}\ c$$

5.1 Constraint-based typing

We now present an algorithm that, when given a typing context Γ and an expression e, produces a set of *constraints*—equations between types (including type variables)—that must be satisfied in order for e to be well-typed in Γ.

We first introduce *type variables*, which are just placeholders for types. We use X and Y to range over type variables.

The language we will consider is the lambda calculus with integer constants and addition. We assume that all function definitions contain a type annotation for the argument, but this type may simply be a type variable X.

$$\begin{aligned} e &::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\ \tau &::= \textbf{int} \mid X \mid \tau_1 \to \tau_2 \end{aligned}$$

To formally define type inference, we introduce a new typing relation:

$$\Gamma \vdash e : \tau \triangleright C$$

Intuitively, if $\Gamma \vdash e : \tau \triangleright C$, then expression e has type τ provided that every constraint in the set C is satisfied. Constraints are of the form $\tau_1 = \tau_2$, intuitively meaning that types τ_1 and τ_2 are equal.

We define the judgment $\Gamma \vdash e : \tau \triangleright C$ with inference rules and axioms. When read from bottom to top, these inference rules provide a procedure that, given Γ and e, calculates τ and C such that $\Gamma \vdash e : \tau \triangleright C$.

$$\text{CT-Var}\ \frac{}{\Gamma \vdash x : \tau \triangleright \varnothing}\ x : \tau \in \Gamma \qquad\qquad \text{CT-Int}\ \frac{}{\Gamma \vdash n : \mathbf{int} \triangleright \varnothing}$$

$$\text{CT-Add}\ \frac{\Gamma \vdash e_1 : \tau_1 \triangleright C_1 \qquad \Gamma \vdash e_2 : \tau_2 \triangleright C_2}{\Gamma \vdash e_1 + e_2 : \mathbf{int} \triangleright C_1 \cup C_2 \cup \{\tau_1 = \mathbf{int}, \tau_2 = \mathbf{int}\}}$$

$$\text{CT-Abs}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \triangleright C}{\Gamma \vdash \lambda x : \tau_1 . e : \tau_1 \to \tau_2 \triangleright C} \qquad \text{CT-App}\ \frac{\Gamma \vdash e_1 : \tau_1 \triangleright C_1 \quad \Gamma \vdash e_2 : \tau_2 \triangleright C_2}{\Gamma \vdash e_1\ e_2 : X \triangleright C_1 \cup C_2 \cup \{\tau_1 = \tau_2 \to X\}}\ X \text{ is fresh}$$

Note that we must be careful with the choice of fresh type variables. We have omitted some of the technical details that ensure the fresh type variables in the rule CT-App are chosen appropriately.

Let's see an example of using this typing judgment to produce a set of constraints. Consider the program $\lambda a : X . \lambda b : Y . 2 + (a\ (b + 3))$.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{a : X, b : Y \vdash 2 : \mathbf{int} \triangleright \varnothing} \qquad \dfrac{\dfrac{}{a : X, b : Y \vdash a : X \triangleright \varnothing} \qquad \dfrac{\dfrac{}{a : X, b : Y \vdash b : Y \triangleright \varnothing} \qquad \dfrac{}{a : X, b : Y \vdash 3 : \mathbf{int} \triangleright \varnothing}}{a : X, b : Y \vdash b + 3 : \mathbf{int} \triangleright \{Y = \mathbf{int}, \mathbf{int} = \mathbf{int}\}}}{a : X, b : Y \vdash a\ (b + 3) : Z \triangleright \{X = \mathbf{int} \to Z, Y = \mathbf{int}, \mathbf{int} = \mathbf{int}\}}}{a : X, b : Y \vdash 2 + (a\ (b + 3)) : \mathbf{int} \triangleright \{Z = \mathbf{int}, X = \mathbf{int} \to Z, Y = \mathbf{int}, \mathbf{int} = \mathbf{int}\}}}{a : X \vdash \lambda b : Y . 2 + (a\ (b + 3)) : Y \to \mathbf{int} \triangleright \{Z = \mathbf{int}, X = \mathbf{int} \to Z, Y = \mathbf{int}, \mathbf{int} = \mathbf{int}\}}}{\vdash \lambda a : X . \lambda b : Y . 2 + (a\ (b + 3)) : X \to Y \to \mathbf{int} \triangleright \{Z = \mathbf{int}, X = \mathbf{int} \to Z, Y = \mathbf{int}, \mathbf{int} = \mathbf{int}\}}$$

The typing derivation means that expression $\lambda a : X . \lambda b : Y . 2 + (a\ (b + 3)) : X \to (Y \to \mathbf{int})$ has type $X \to Y \to \mathbf{int}$ provided that we can satisfy the constraints $Z = \mathbf{int}, X = \mathbf{int} \to Z, Y = \mathbf{int}$, and $\mathbf{int} = \mathbf{int}$.

5.2 Unification

So what does it mean for a set of constraints to be satisfied? To answer this question, we define *type substitutions* (or just *substitutions*, when it's clear from context).

5.2.1 Type substitution

A type substitution is a finite map from type variables to types. For example, we write $[X \mapsto \mathbf{int}, Y \mapsto \mathbf{int} \to \mathbf{int}]$ for the substitution that maps type variable X to $\mathbf{int}$, and type variable Y to $\mathbf{int} \to \mathbf{int}$.

Note that the same variable could occur in both the domain and range of a substitution. In that case, the intention is that all substitutions are performed simultaneously. For example the substitution $[X \mapsto \mathbf{int}, Y \mapsto \mathbf{int} \to X]$ maps Y to $\mathbf{int} \to \mathbf{X}$ (not to $\mathbf{int} \to \mathbf{int}$).

More formally, we define substitution of type variables as follows.

$$\sigma(X) = \begin{cases} \sigma(\tau) & \text{if } X \mapsto \tau \in \sigma \\ X & \text{if } X \text{ not in the domain of } \sigma \end{cases}$$
$$\sigma(\textsf{int}) = \textsf{int}$$
$$\sigma(\tau \to \tau') = \sigma(\tau) \to \sigma(\tau')$$

(This is only well defined if σ is acyclic, that is, does not require the substitution of variable X within the result of substituting variable X.)

Note that we don't need to worry about avoiding variable capture, since there are no constructs in the language that bind type variables. (We'll soon see more sophisticated typing constructs that introduce binders for type variables.)

We can extend substitution to constraints, and sets of constraints in the obvious way:

$$\sigma(\tau_1 = \tau_2) = \sigma(\tau_1) = \sigma(\tau_2)$$
$$\sigma(C) = \{\sigma(c) \mid c \in C\}$$

Given two substitutions σ and σ', we write $\sigma \circ \sigma'$ for the composition of the substitutions: $\sigma \circ \sigma'(\tau) = \sigma(\sigma'(\tau))$.

5.2.2 Unification

Constraints are of the form $\tau = \tau'$. We say that a substitution σ *unifies* constraint $\tau = \tau'$ if $\sigma(\tau)$ is the same as $\sigma(\tau')$. We say that substitution σ *satisfies* (or *unifies*) set of constraints C if σ unifies every constraint in C.

For example, the substitution $\sigma = [X \mapsto \textsf{int}, Y \mapsto \textsf{int} \to \textsf{int}]$ unifies the constraint $X \to (X \to \textsf{int}) = \textsf{int} \to Y$, since

$$\sigma(X \to (X \to \textsf{int})) \quad = \quad \textsf{int} \to (\textsf{int} \to \textsf{int}) \quad = \quad \sigma(\textsf{int} \to Y)$$

So to solve a set of constraints C, we need to find a substitution that unifies C. More specifically, suppose that $\Gamma \vdash e : \tau \triangleright C$. Expression e is typable if and only if there is a substitution σ that satisfies C, and moreover, the type of e is $\sigma(\tau)$. If there are no substitutions that satisfy C, then we know that e is not typable.

5.2.3 Unification algorithm

To calculate solutions to constraint sets, we use the idea, due to Hindley and Milner, of using *unification* to check that the set of solutions is non-empty, and to find a "best" solution (from which all other solutions can be easily generated).

The algorithm for unification is defined as follows.

$$\begin{aligned}
\mathit{unify}(\varnothing) &= [] \qquad \text{(the empty substitution)} \\
\mathit{unify}(\{\tau = \tau'\} \cup C') &= \text{if } \tau = \tau' \text{ then} \\
&\qquad \mathit{unify}(C') \\
&\quad \text{else if } \tau = X \text{ and } X \text{ not a free variable of } \tau' \text{ then} \\
&\qquad \mathit{unify}([X \mapsto \tau'](C')) \circ [X \mapsto \tau'] \\
&\quad \text{else if } \tau' = X \text{ and } X \text{ not a free variable of } \tau \text{ then} \\
&\qquad \mathit{unify}([X \mapsto \tau](C')) \circ [X \mapsto \tau] \\
&\quad \text{else if } \tau = \tau_0 \to \tau_1 \text{ and } \tau' = \tau_0' \to \tau_1' \text{ then} \\
&\qquad \mathit{unify}(C' \cup \{\tau_0 = \tau_0', \tau_1 = \tau_1'\}) \\
&\quad \text{else} \\
&\qquad \mathit{fail}
\end{aligned}$$

The check that X is not a free variable of the other type ensures that the algorithm doesn't produce a cyclic substitution (e.g., $X \mapsto (X \to X)$), which doesn't make sense with the finite types that we currently have.

The unification algorithm always terminates. (Hint: what decreases on every recursive invocation? this is a bit tricky because one of the cases *adds* constraints to the set of constraints!) Moreover, it produces a solution if and only if a solution exists. The solution found is the most general solution, in the sense that if $\sigma = \mathit{unify}(C)$ and σ' is a solution to C, then there is some σ'' such that $\sigma' = \sigma'' \circ \sigma$.

6 Parametric polymorphism

Polymorph means "many forms". *Polymorphism* is the ability of code to be used on values of different types. For example, a polymorphic function is one that can be invoked with arguments of different types. A polymorphic datatype is one that can contain elements of different types.

Several kinds of polymorphism are commonly used in modern languages.

- *Subtype polymorphism* gives a single term many types using the subsumption rule. For example, a function with argument τ can operate on any value with a type that is a subtype of τ.
- *Ad-hoc polymorphism* usually refers to code that appears to be polymorphic to the programmer, but the actual implementation is not. A typical example is *overloading*: using the same function name for functions with different kinds of parameters. Although it looks like a polymorphic function to the code that uses it, there are actually multiple function implementations (none being polymorphic) and the compiler invokes the appropriate one. Ad-hoc polymorphism is a dispatch mechanism: the type of the arguments is used to determine (either at compile time or run time) which code to invoke.
- *Parametric polymorphism* refers to code that is written without knowledge of the actual type of the arguments; the code is parametric in the type of the parameters. Examples include polymorphic functions in ML, or generics in Java 5.

We consider parametric polymorphism in more detail. Suppose we are working in the simply-typed lambda calculus, and consider an "apply twice" function for integers that takes a function f, and an integer x, applies f to x, and then applies f to the result.

$$\mathit{appTwiceInt} \triangleq \lambda f{:}\mathsf{int} \to \mathsf{int}.\, \lambda x{:}\mathsf{int}.\, f\ (f\ x)$$

We could also write an apply-twice function for booleans. Or for functions over integers. Or for any other type...

$$\begin{aligned} \mathit{appTwiceBool} &\triangleq \lambda f{:}\mathsf{bool} \to \mathsf{bool}.\, \lambda x{:}\mathsf{bool}.\, f\ (f\ x) \\ \mathit{appTwiceFn} &\triangleq \lambda f{:}(\mathsf{int} \to \mathsf{int}) \to (\mathsf{int} \to \mathsf{int}).\, \lambda x{:}\mathsf{int} \to \mathsf{int}.\, f\ (f\ x) \\ &\vdots \end{aligned}$$

In the simply typed lambda calculus, if we want to apply the doubling operation to different types of arguments in the same program, we need to write a new function for each type. This violates the *abstraction principle* of software engineering:

> Each significant piece of functionality in a program should be implemented in just one place in the source code. When similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

In the doubling functions above, the varying parts are the types. We need a way to abstract out the type of the doubling operation, and later instantiate this abstract type with different concrete types.

We extend the simply-typed lambda calculus with abstraction over types, giving the *polymorphic lambda calculus*, also called *System F*.

A *type abstraction* is a new expression, written $\Lambda X.\, e$, where Λ is the upper-case form of the Greek letter lambda, and X is a *type variable*. We also introduce a new form of application, called *type application*, or *instantiation*, written $e_1\ [\tau]$.

When a type abstraction meets a type application during evaluation, we substitute the free occurrences of the type variable with the type. Note that instantiation does not require the program to keep run-time type information, or to perform type checks at run-time; it is just used as a way to statically check type safety in the presence of polymorphism.

6.1 Syntax and operational semantics

The new syntax of the language is given by the following grammar.

$$e ::= n \mid x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \Lambda X.\, e \mid e\ [\tau]$$
$$v ::= n \mid \lambda x{:}\tau.\, e \mid \Lambda X.\, e$$

The evaluation rules for the polymorphic lambda calculus are the same as for the simply-typed lambda calculus, augmented with new rules for evaluating type application.

$$E ::= [\cdot] \mid E\ e \mid v\ E \mid E\ [\tau]$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \beta\text{-reduction}\ \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e\{v/x\}}$$

$$\textsc{Type-reduction}\ \frac{}{(\Lambda X.\, e)\ [\tau] \longrightarrow e\{\tau/X\}}$$

Let's consider an example. In this language, the polymorphic identity function is written as

$$ID \triangleq \Lambda X.\, \lambda x{:}X.\, x$$

We can apply the polymorphic identity function to **int**, producing the identity function on integers.

$$(\Lambda X.\, \lambda x{:}X.\, x)\ [\mathsf{int}] \longrightarrow \lambda x{:}\mathsf{int}.\, x$$

We can apply *ID* to other types as easily:

$$(\Lambda X.\, \lambda x{:}X.\, x)\ [\mathsf{int} \to \mathsf{int}] \longrightarrow \lambda x{:}\mathsf{int} \to \mathsf{int}.\, x$$

6.2 Type system

We also need to provide a type for the new type abstraction. The type of $\Lambda X.\, e$ is $\forall X.\, \tau$, where τ is the type of e, and may contain the type variable X. Intuitively, we use this notation because we can instantiate the type expression with any type for X: for any type X, expression e can have the type τ (which may mention X).

$$\tau ::= \mathsf{int} \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\, \tau$$

Type checking expressions is slightly different than before. Besides the type environment Γ (which maps variables to types), we also need to keep track of the set of type variables Δ. This is to ensure that a type variable X is only used in the scope of an enclosing type abstraction $\Lambda X.\, e$. Thus, typing judgments are now of the form $\Delta; \Gamma \vdash e : \tau$, where Δ is a set of type variables, and Γ is a typing context. We also use an additional judgment $\Delta \vdash \tau\ \mathsf{ok}$ to ensure that type τ uses only type variables from the set Δ.

$$\frac{}{\Delta;\Gamma \vdash n : \mathbf{int}} \qquad \frac{\Delta \vdash \tau \text{ ok}}{\Delta;\Gamma \vdash x : \tau}\,\Gamma(x) = \tau \qquad \frac{\Delta;\Gamma, x:\tau \vdash e:\tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta;\Gamma \vdash \lambda x{:}\tau.\, e : \tau \to \tau'}$$

$$\frac{\Delta;\Gamma \vdash e_1 : \tau \to \tau' \quad \Delta;\Gamma \vdash e_2 : \tau}{\Delta;\Gamma \vdash e_1\ e_2 : \tau'} \qquad \frac{\Delta, X;\Gamma \vdash e:\tau}{\Delta;\Gamma \vdash \Lambda X.\, e : \forall X.\ \tau} \qquad \frac{\Delta;\Gamma \vdash e : \forall X.\ \tau' \quad \Delta \vdash \tau \text{ ok}}{\Delta;\Gamma \vdash e\ [\tau] : \tau'\{\tau/X\}}$$

$$\frac{}{\Delta \vdash X \text{ ok}}\,X \in \Delta \qquad \frac{}{\Delta \vdash \mathbf{int} \text{ ok}} \qquad \frac{\Delta \vdash \tau_1 \text{ ok} \quad \Delta \vdash \tau_2 \text{ ok}}{\Delta \vdash \tau_1 \to \tau_2 \text{ ok}} \qquad \frac{\Delta \cup \{X\} \vdash \tau \text{ ok}}{\Delta \vdash \forall X.\ \tau \text{ ok}}$$

6.3 Examples

Let's consider the apply-twice operation again. We can write a polymorphic apply-twice operation as

$$\mathit{appTwice} \triangleq \Lambda X.\, \lambda f{:}X \to X.\, \lambda x{:}X.\, f\ (f\ x).$$

The type of this expression is

$$\forall X.\ (X \to X) \to X \to X$$

We can instantiate this on a type, and provide arguments. For example,

$$\begin{aligned} \mathit{appTwice}\ [\mathbf{int}]\ (\lambda n{:}\mathbf{int}.\, n+1)\ 7 &\longrightarrow (\lambda f{:}\mathbf{int} \to \mathbf{int}.\, \lambda x{:}\mathbf{int}.\, f\ (f\ x))\ (\lambda n{:}\mathbf{int}.\, n+1)\ 7 \\ &\longrightarrow^* 9 \end{aligned}$$

Recall that in the simply-typed lambda calculus, we had no way of typing the expression $\lambda x.\, x\ x$. In the polymorphic lambda calculus, however, we can type this expression if we give it a polymorphic type and instantiate it appropriately.

$$\vdash \quad \lambda x{:}\forall X.\ X \to X.\, x\ [\forall X.\ X \to X]\ x \quad : \quad (\forall X.\ X \to X) \to (\forall X.\ X \to X)$$

6.4 Erasure

The semantics of System F presented above explicitly passes type. In an implementation, one often wants to eliminate types for efficiency. The following translation "erases" the types from a System F expression.

$$\begin{aligned} \mathit{erase}(x) &= x \\ \mathit{erase}(n) &= n \\ \mathit{erase}(\lambda x{:}\tau.\, e) &= \lambda x.\, \mathit{erase}(e) \\ \mathit{erase}(e_1\ e_2) &= \mathit{erase}(e_1)\ \mathit{erase}(e_2) \\ \mathit{erase}(\Lambda X.\, e) &= \lambda z.\, \mathit{erase}(e) && \text{where } z \notin FV(e) \\ \mathit{erase}(e\ [\tau]) &= \mathit{erase}(e)(\lambda x.\, x) \end{aligned}$$

The following theorem states that the translation is adequate.

Theorem 4 (Adequacy)**.** *For all expressions e and e', we have $e \longrightarrow^* e'$ iff $\mathit{erase}(e) \longrightarrow^* \mathit{erase}(e')$.*

The type reconstruction problem asks whether, for a given untyped λ-calculus expression e' there exists a well-typed System F expression e such that $\mathit{erase}(e) = e'$. It was shown to be undecidable by Wells in 1994, by showing at type checking is undecidable for a variant of untyped λ-calculus without annotations. See Pierce Chapter 23 for further discussion, and restrictions of System F for which type reconstruction is decidable.

In our language, we have explicit annotations for type abstraction ($\Lambda X.\, e$) and type application ($e\ [\tau]$). Given these annotations, then type checking is decidable, and an adaptation of our constraint generating type inference system would work.

But in real languages such as ML, programmers don't have to annotate their programs with $\forall X.\ \tau$ or $e\ [\tau]$. Both are automatically inferred by the compiler (although the programmer can specify the former if he wishes). For example, we can write `let double f x = f (f x);;` and Ocaml will figure out that the type is `('a -> 'a) -> 'a -> 'a` (which is roughly equivalent to $\forall A.\ (A \to A) \to A \to A$).

We can also write `appTwice (fun x -> x+1) 7;;`, and Ocaml will infer that the polymorphic function `appTwice` is instantiated on the type `int`.

So what's going on? How can type inference in these languages work?

The polymorphism in ML is not exactly like the polymorphism in System F. ML restricts what types a type variable may be instantiated with. Specifically, type variables can not be instantiated with polymorphic types. Also, polymorphic types are not allowed to appear on the left-hand side of arrows (i.e., a polymorphic type cannot be the type of a function argument). This form of polymorphism is known as *let-polymorphism* (due to the special role played by let in ML), or *prenex polymorphism*. These restrictions ensure that *type inference* is possible.

An example of a term that is typable in System F but not typable in ML is the self-application expression $\lambda x.\, x\ x$. It can be typed in System F as

$$\vdash \lambda x : \forall X.\ X \to X.\, x\ [\forall X.\ X \to X]\ x : (\forall X.\ X \to X) \to (\forall X.\ X \to X)$$

7 Records

We have previously seen binary products, i.e., pairs of values. Binary products can be generalized in a straightforward way to n-ary products, also called *tuples*. For example, $\langle 3, (), \mathsf{true}, 42\rangle$ is a 4-ary tuple containing an integer, a unit value, a boolean value, and another integer. Its type is $\mathbf{int} \times \mathbf{unit} \times \mathbf{bool} \times \mathbf{int}$.

Records are a generalization of tuples. We annotate each field of record with a *label*, drawn from some set of labels $\mathcal{L}$. For example, $\{\mathsf{foo} = 32, \mathsf{bar} = \mathsf{true}\}$ is a record value with an integer field labeled foo and a boolean field labeled bar. The type of the record value is written $\{\mathsf{foo} : \mathbf{int}, \mathsf{bar} : \mathbf{bool}\}$.

We extend the syntax, operational semantics, and typing rules of the call-by-value lambda calculus to support records.

$$\begin{aligned} & l \in \mathcal{L} \\ & e ::= \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\ & v ::= \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\ & \tau ::= \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \end{aligned}$$

We add new evaluation contexts to evaluate the fields of records.

$$E ::= \cdots \mid \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\} \mid E.l$$

We also add a rule to access the field of a record.

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \longrightarrow v_i}$$

Finally, we add new typing rules for records.

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

Note that the order of labels is important: the type of the record value $\{\mathsf{lat} = -40, \mathsf{long} = 175\}$ is $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$, which is different from $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$, the type of the record value $\{\mathsf{long} = 175, \mathsf{lat} = -40\}$. In many languages with records, the order of the labels is not important.

There are several ways in which we could make the order of labels irrelevant. Each comes at the cost of some complication somewhere in the type system.

- We could replace the typing rules for records with one that allows a different order of labels in the record and in the type. For instance

$$\frac{\Gamma \vdash e_1 : \tau_{\pi(1)} \quad \ldots \quad \Gamma \vdash e_n : \tau_{\pi(n)}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l'_1 : \tau_1, \ldots, l'_n : \tau_n\}} \; \pi \text{ a permutation of } \{1, \ldots, n\},\ l_i = l_{\pi(i)} \text{ for all } i$$

  The cost here is that no longer have a unique type for every expression. A record expression can be typed at multiple types.

- We could consider the types $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$ and $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$ to be equal, by introducing an equivalence relation over types. The cost here is that all the typing rules need to take this equivalence relation into account: a function annotated to accept an argumnet of type τ and given an argument of type τ' should type check if τ and τ' are equivalent types. This is often handled by introducing a new judgment for determining type equivalence.

- We could rely on subtyping — we will see this approach next time. The cost here is that once again we have expressions with multiple types.

- We could simply avoid the problem and require that the labels in a record and in a type be kept in a canonical order as a matter of syntax: if $<$ is a linear order over $\mathcal{L}$, we simply require that records $\{l_1 = e_1, \ldots, l_n = e_n\}$ and a record types $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$ satisfy $l_1 < \ldots < l_n$. The cost here is that the language is more restrictive than might be desired. Then again, when the lambda calculus is used as an intermediate language for compilation or interpretation, those kind of restrictions can be enforced by the parser which simply orders the labels in the requisite canonical form.

8 Existential types

We can extend the simply-typed lambda calculus with the dual to universal types: *existential types* (and records). An existential type is written $\exists X.\ \tau$, where type variable X may occur in τ. If a value has type $\exists X.\ \tau$, it means that it is a pair $\{\tau', v\}$ of a type τ' and a value v, such that v has type $\tau\{\tau'/X\}$.

Thinking about constructive logic may provide some intuition for existential types. As the notation and name suggest, the logical formula that corresponds to an existential type $\exists X.\ \tau$ is an existential formula $\exists X.\ \varphi$, where X may occur in φ. In constructive logic, what would it mean for the statement "there exists some X such that φ is true" to be true? In constructive logic, a statement is true only if there is a proof for it. To prove "there exists some X such that φ is true" we must actually provide a *witness* ψ, an entity that is a suitable replacement for X, and also, a proof that φ is true when we replace X with witness ψ.

A value $\{\tau', v\}$ of type $\exists X.\ \tau$ exactly corresponds to a proof of an existential statement: type τ' is the witness type, and v is a value with type $\tau\{\tau'/X\}$.

We introduce a language construct to create existential values, and a construct to use existential values. The syntax of the new language is given by the following grammar.

$$\begin{aligned}
e &::= x \mid \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid n \mid e_1 + e_2 \\
&\quad \mid \{\ l_1 = e_1, \ldots, l_n = e_n\ \} \mid e.l \\
&\quad \mid \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 \\
v &::= n \mid \lambda x{:}\tau.\, e \mid \{\ l_1 = v_1, \ldots, l_n = v_n\ \} \mid \mathsf{pack}\ \{\tau_1, v\}\ \mathsf{as}\ \exists X.\ \tau_2 \\
\tau &::= \mathbf{int} \mid \tau_1 \to \tau_2 \mid \{\ l_1{:}\tau_1, \ldots, l_n{:}\tau_n\ \} \mid X \mid \exists X.\ \tau
\end{aligned}$$

Note that in this grammar, we annotate existential values with their existential type. The construct to create an existential value, $\mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2$, is often called *packing*, and the construct to use an existential value is called *unpacking*.

Before we present the operational semantics and typing rules, let's see some examples to get an intuition for packing and unpacking. Existential types provide us with a mechanism to reason about *modules* which can hide their implementation details. That is, a module that wants to hide away its internal details tells

the external world that there is *some* type or types that describe its internal structures and implementation, but the clients are not allowed to know anything about these implementation types, simply that they exist.

Here we create an existential value that implements a counter, without revealing details of its implementation.

$$\begin{array}{l}\mathsf{let}\ \mathit{counterADT} = \\ \quad \mathsf{pack} \\ \quad\quad \{\mathbf{int}, \{\ \mathsf{new} = 0, \mathsf{get} = \lambda i{:}\mathbf{int}.\, i, \mathsf{inc} = \lambda i{:}\mathbf{int}.\, i+1\ \}\ \} \\ \quad \mathsf{as} \\ \quad\quad \exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \} \\ \mathsf{in}\ \ldots\end{array}$$

The abstract type name is **Counter**, and its concrete representation is **int**. The type of the variable *counterADT* is $\exists \mathbf{Counter}.\ \{\ \mathsf{new} : \mathbf{Counter}, \mathsf{get} : \mathbf{Counter} \to \mathbf{int}, \mathsf{inc} : \mathbf{Counter} \to \mathbf{Counter}\ \}$.

We can use the existential value *counterADT* as follows.

$$\mathsf{unpack}\ \{C, x\} = \mathit{counterADT}\ \mathsf{in}\ \mathsf{let}\ y{:}C = x.\mathsf{new}\ \mathsf{in}\ x.\mathsf{get}\ (x.\mathsf{inc}\ (x.\mathsf{inc}\ y))$$

Note that we annotate the pack construct with the existential type. That is, we explicitly state the type $\exists \mathbf{Counter}.\ \ldots$. Why is this? Without this annotation, we would not know which occurrences of the witness type are intended to be replaced with the type variable, and which are intended to be left as the witness type. In the counter example above, the type of expressions $\lambda i{:}\mathbf{int}.\, i$ and $\lambda i{:}\mathbf{int}.\, i+1$ are both $\mathbf{int} \to \mathbf{int}$, but one is the implementation of get, of type $\mathbf{Counter} \to \mathbf{int}$ and the other is the implementation of inc, of type $\mathbf{Counter} \to \mathbf{Counter}$.

We now define the operational semantics. We add two new evaluation contexts, and one evaluation rule for unpacking an existential value.

$$E ::= \cdots \mid \mathsf{pack}\ \{\tau_1, E\}\ \mathsf{as}\ \exists X.\ \tau_2 \mid \mathsf{unpack}\ \{X, x\} = E\ \mathsf{in}\ e$$

$$\frac{}{\mathsf{unpack}\ \{X, x\} = (\mathsf{pack}\ \{\tau_1, v\}\ \mathsf{as}\ \exists Y.\ \tau_2)\ \mathsf{in}\ e \longrightarrow e\{v/x\}\{\tau_1/X\}}$$

The new typing rules make sure that existential values are used correctly. Note that code using an existential value (e in $\mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2$) does not know the witness type of the existential value of type $\exists X.\ \tau_1$.

$$\frac{\Delta; \Gamma \vdash e{:}\tau_2\{\tau_1/X\}}{\Delta; \Gamma \vdash \mathsf{pack}\ \{\tau_1, e\}\ \mathsf{as}\ \exists X.\ \tau_2 {:} \exists X.\ \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e_1{:}\exists X.\ \tau_1 \quad X \notin \Delta \quad \Delta, X; \Gamma, x{:}\tau_1 \vdash e_2{:}\tau_2 \quad \Delta \vdash \tau_2\ \mathsf{ok}}{\Delta; \Gamma \vdash \mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2 {:} \tau_2} \qquad \frac{\Delta, X \vdash \tau\ \mathsf{ok}}{\Delta \vdash \exists X.\ \tau\ \mathsf{ok}}$$

The judgments have the same form as they do for universal types, including a well-formedness judgment $\Delta \vdash \tau\ \mathsf{ok}$. We define well-formedness of existential types, similar to well-formedness of universal types. In the typing rule for $\mathsf{unpack}\ \{X, x\} = e_1\ \mathsf{in}\ e_2$, note that we have the premises $X \notin \Delta$ and $\Delta \vdash \tau_2\ \mathsf{ok}$. The first ensures that X is not currently a type variable in scope (and we can alpha-vary it to ensure that this holds true). Why do we need the premise $\Delta \vdash \tau_2\ \mathsf{ok}$?

9 Subtyping

Subtyping is a key feature of object-oriented languages. Subtyping was first introduced in SIMULA, invented by Norwegian researchers Dahl and Nygaard, and considered the first object-oriented programming language.

The principle of subtyping is as follows. If τ_1 is a subtype of τ_2 (written $\tau_1 <: \tau_2$, and also sometimes as $\tau_1 \leq: \tau_2$), then a program can use a value of type τ_1 whenever it would use a value of type τ_2. If $\tau_1 <: \tau_2$, then τ_1 is sometimes referred to as the subtype, and τ_2 as the supertype.

The goal here is to type more expressions, while still retaining type soundness. To motivate this idea, consider the function

$$\lambda x{:}\{\mathsf{a}:\mathbf{int},\mathsf{b}:\mathbf{int}\}.\, x.\mathsf{a} + x.\mathsf{b}.$$

Clearly, we can give it record $\{\mathsf{a} = 10, \mathsf{b} = 20\}$ as an argument. But it's also fine (as in: the resulting expression does not get stuck) if we give it record $\{\mathsf{a} = 10, \mathsf{b} = 20, \mathsf{c} = 30\}$. But the type system does not allow it.

We can express the principle of subtyping in a typing rule, often referred to as the "subsumption typing rule" (since the supertype subsumes the subtype).

$$\textsc{Subsumption}\ \frac{\Gamma \vdash e{:}\tau \quad \vdash \tau <: \tau'}{\Gamma \vdash e{:}\tau'}$$

The subsumption rule says that if e is of type τ, and τ is a subtype of τ', then e is also of type τ'.

Recall that we provided an intuition for a type as a set of computational entities that share some common property. Type τ is a subtype of type τ' if every computational entity in the set for τ can be regarded as a computational entity in the set for τ'.

So what types are in a subtype relation? We will define inference rules and axioms for the subtype relation $<:$, via a judgment $\vdash \tau <: \tau'$.

The subtype relation is both reflexive and transitive. These properties both seem reasonable if we think of subtyping as a subset relation. We add inference rules that express this.

$$\frac{}{\vdash \tau <: \tau} \qquad\qquad \frac{\vdash \tau_1 <: \tau_2 \quad \vdash \tau_2 <: \tau_3}{\vdash \tau_1 <: \tau_3}$$

9.1 Subtyping for records

Consider records and record types. A record consists of a set of labeled fields. Its type includes the types of the fields in the record. Let's define the type **Point** to be the record type $\{\mathsf{x}:\mathbf{int},\mathsf{y}:\mathbf{int}\}$, that contains two fields x and y, both integers. That is:

$$\mathbf{Point} = \{\mathsf{x}:\mathbf{int},\mathsf{y}:\mathbf{int}\}.$$

Lets also define

$$\mathbf{Point3D} = \{\mathsf{x}:\mathbf{int},\mathsf{y}:\mathbf{int},\mathsf{z}:\mathbf{int}\}$$

as the type of a record with three integer fields x, y and z.

Because **Point3D** contains all of the fields of **Point**, and those have the same type as in **Point**, it makes sense to say that **Point3D** is a subtype of **Point**: **Point3D** $<:$ **Point**.

Think about any code that used a value of type **Point**. This code could access the fields x and y, and that's pretty much all it could do with a value of type **Point**. A value of type **Point3D** has these same fields, x and y, and so any piece of code that used a value of type **Point** could instead use a value of type **Point3D**.

We can write a subtyping rule for records.

$$\frac{}{\vdash \{l_1{:}\tau_1, \ldots, l_{n+k}{:}\tau_{n+k}\} <: \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}}\ k \geq 0$$

But why not let the corresponding fields be in a subtyping relation? For example, if $\tau_1 \leq \tau_2$ and $\tau_3 \leq \tau_4$, then is $\{\mathsf{foo} : \tau_1, \mathsf{bar} : \tau_3\}$ a subtype of $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$? Turns out that this is the case so long as the fields of records are immutable. More on this when we consider subtyping for references.

Also, we could relax the requirement that the order of fields must be the same. The following is a more permissive subtyping rule for records.

$$\frac{\vdash \tau_{f(1)} <: \tau'_1 \quad \ldots \quad \vdash \tau_{f(m)} <: \tau'_m}{\vdash \{l_1{:}\tau_1, \ldots, l_m{:}\tau_m\} <: \{l'_1{:}\tau'_1, \ldots, l'_n{:}\tau'_n\}}\ f : \{1,\ldots,n\} \to \{1,\ldots,m\}, l'_i = l_{f(i)}$$

9.2 Subtyping for functions

Consider two function types $\tau_1 \to \tau_2$ and $\tau_1' \to \tau_2'$. What are the subtyping relations between τ_1,, τ_2, τ_1',, and τ_2' that should be satisfied in order for $\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$ to hold?

Consider the following expression:

$$G \triangleq \lambda f{:}\tau_1' \to \tau_2'.\, \lambda x{:}\tau_1'.\, f\ x.$$

This function has type

$$(\tau_1' \to \tau_2') \to \tau_1' \to \tau_2'.$$

Now suppose we had a function $h{:}\tau_1 \to \tau_2$ such that $\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$. By the subtyping principle, we should be able to give h as an argument to G, and G should work fine. Suppose that v is a value of type τ_1'. Then $G\ h\ v$ will evaluate to $h\ v$, meaning that h will be passed a value of type τ_1. Since h has type $\tau_1 \to \tau_2$, it must be the case that $\tau_1' <: \tau_1$. (What could go wrong if $\tau_1 <: \tau_1'$?)

Furthermore, the result type of $G\ h\ v$ should be of type τ_2' according to the type of G, but $h\ v$ will produce a value of type τ_2, as indicated by the type of h. So it must be the case that $\tau_2 <: \tau_2'$.

Putting these two pieces together, we get the typing rule for function types.

$$\frac{\vdash \tau_1' <: \tau_1 \quad \vdash \tau_2 <: \tau_2'}{\vdash \tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Note that the subtyping relation between the argument and result types in the premise are in different directions! The subtype relation for the result type is in the same direction as for the conclusion (primed version is the supertype, non-primed version is the subtype); it is in the opposite direction for the argument type. We say that subtyping for the function type is *covariant* in the result type, and *contravariant* in the argument type.

9.3 Subtyping for products and sums

Like records, we can allow the elements of a product to be in a subtyping relation.

$$\frac{\vdash \tau_1 <: \tau_1' \quad \vdash \tau_2 <: \tau_2'}{\vdash \tau_1 \times \tau_2 <: \tau_1' \times \tau_2'}$$

Similarly, we can allow the elements of a sum to be in a subtyping relation.

$$\frac{\vdash \tau_1 <: \tau_1' \quad \vdash \tau_2 <: \tau_2'}{\vdash \tau_1 + \tau_2 <: \tau_1' + \tau_2'}$$