Background on type systems

A type error occurs when we attempt to do some primitive operation (such as function application) but the operation is not defined for the given operands. For instance, there is no transition rule for applying an integer to an integer, so if we ever get into a situation where we are doing this, we have a type error:

$$(\lambda f.\, f\ 42)\ 3 \rightarrow 3\ 42 \qquad \text{stuck!}$$

In the pure, untyped lambda calculus, the only built-in operation we had was function application, and the only values we had were functions. Thus, for a (closed) expression, there was no way to get a type error.

As soon as we add something besides functions, we have the potential for type errors. We say that a language is **type safe** if there is no well-formed program that can get stuck due to a type error.

The goal of a type system is to determine whether or not a given expression will ever evaluate to a configuration that is stuck due to a type error, and if so, rule out that expression from the language. Note that in general, it is undecidable whether or not a program can get stuck (i.e., you can solve the halting problem if you claim you have a perfect type checker). So, all static type checkers are conservative.

An alternative to static type checking is dynamic type checking. Dynamic type checking augments the semantics so that, no matter what primitive operation we're doing, and no matter what values we have, we always have a transition we can take. For instance, in Scheme, if you attempt to apply 3 to 42, then this transitions to a special error value:

$$(\lambda f.\, f\ 42)\ 3 \rightarrow 3\ 42 \rightarrow \mathsf{error}$$

Similarly, if you attempt add error to a number, you get the error value, and so forth. Dynamic type checking has its benefits, because it does not suffer from the conservative aspects of static typing. However, dynamic type checking has two drawbacks: First, it requires run-time type information and run-time checks for primitive operations. For example, in Scheme, when you see v_1 applied to v_2, you have to check to see whether or not v1 is a function, and if so, then use the normal function-call semantics. Otherwise, you have to return the error value. Thus, we need to be able to tell at run-time whether or not a value is a function. Similarly, if you attempt to add two values, we have to dynamically check that the two values are numbers, and if not, return the error value. So, we must be able to tell numbers from all other values.

The second problem with dynamic type checking is that you must exhaustively test your program to rule out type errors. Modern languages make it easy to write statically-typed programs (i.e., the type systems are not so conservative that they rule out lots of useful programs) so static type checking is a good way to test, once and for all, that your program will not have a type error when run. From a software engineering standpoint, this is a crucial advantage as it allows you to concentrate on real bugs, instead of silly little type errors. (And if you think that type errors don't really happen in practice for dynamically typed languages, you are either very foolish or omnipotent.)

In practice, all type-safe languages use a mix of static and dynamic typing. For instance, ML is mostly statically typed but some things are checked dynamically, including division by zero, and array bounds. This is because a simple analysis won't easily be able to prove integer constraints (e.g., $x \neq 0$ or $0 \leq x < \mathsf{size}(A)$.) That's not to say that we couldn't devise a type system which statically enforces these things, rather that the type systems tend to become complicated. When designing a language, you want to keep your type system simple, yet expressive so that you can rule out most type errors without making the programmer mad at you.

Java is also mostly statically typed, but unfortunately, it requires many more (implicit) run-time checks than ML. For instance, in addition to divide by zero and array subscripts, Java implementations must check that when you write into an array, then the type of the object is equal to the element type of the array. As we'll see, a slight change to the Java type system would've made this check unnecessary. As another example, Java implementations must check for null pointers on method call or member access, and they must check a sub-typing relation on a downcast.

More expressive type systems can often avoid run-time checks. For instance, in ML, by default, "pointers" cannot be null. You can code up "pointers that might be null" using a datatype:

```
datatype 'a option = None | Some of 'a
```

In Java, you can't even express a non-null reference. So, you're forced to do a lot of testing that's really unnecessary. There are proposals to strengthen Java's type system to support "not-null" references. Surprisingly, if we chose "not-null" as the default, most code would type-check as is! This was a real bug in the Java design as far as I'm concerned.

Similarly, downcasts are not needed nearly as often if you have some form of parametric polymorphism (as in ML). Here, Sun has finally gotten its act together and incorporated polymorphism (a.k.a. generics) in the next release of Java. Sadly, they didn't fix the array update problem which they could've (generics solve this nicely.)

But ML isn't without its faults. There's no subtyping in ML, as there is in Java. So, in practice, you're often forced to duplicate code in ML for different types, even though it's the same code. (Actually, you can often encode the subtyping with polymorphism, but not always.) Duplicating code is a bad thing because if there's a bug in one copy, it's likely there's a bug in the other copy. Testing might only reveal one bug, so you'll have to remember to hunt down all of the other copies and make the bug fixes there too. Or, if you go to performance-tune the code, you'll have to do it in multiple places.

A key principle of language design is that the language should provide enough power to abstract out common bits of code so that you only have to write them once. This simplifies testing, debugging, and tends to lead to smaller and more robust code. A restrictive type system (as say in Pascal) prevents code sharing, which is why some people still think that dynamic typing is the best thing. But with the right type structure, you can usually avoid code duplication without having to do exhaustive testing.

Proving the soundness of the simply-typed lambda calculus

Here's the simply-typed lambda calculus:

types	$t ::= b \mid \tau_1 \to \tau_2$
expressions	$e ::= c \mid x \mid \lambda x.\, e \mid e_1\ e_2$
values	$v ::= c \mid \lambda x.\, e$

where we have some set of constants (c) described by a base type (b). Think c is drawn from $\{1, 2, 3, \ldots\}$ and $b = \mathsf{int}$ if you like.

The (call-by-value) operational semantics is given by:

$$(\lambda x.\, e)\ v \to e[v/x]$$

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$$

$$\frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

Recall that substitution $e_1[e_2/x]$ is defined by:

$$\begin{array}{rcll} x[e_2/x] & = & e_2 & \\ y[e_2/x] & = & y & (y \neq x) \\ c[e_2/x] & = & c & \\ (e\ e')[e_2/x] & = & (e[e_2/x])\ (e'[e_2/x]) & \\ (\lambda x.\, e)[e_2/x] & = & \lambda x.\, e & \\ (\lambda y.\, e)[e_2/x] & = & \lambda y.\, (e[e_2/x]) & \end{array}$$

(We are only substituting closed terms so we don't have to worry about capture in the last case.)

The following rules define when, under a set of typing assumptions Γ, a given expression e has type τ (written $\Gamma \vdash e : \tau$). Recall that our typing assumption Γ is a partial function from variables to types. We can think of Γ as the "symbol table" in a compiler that is recording which variables are in scope, and what their types are.

(const) $\Gamma \vdash c : b$ (integers have type int)

(var) $\Gamma \vdash x : \Gamma(x)$ (look up x in the symbol table)

(lam) $$\frac{\Gamma[x \mapsto \tau_1] \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2}$$ (assume x has type τ_1, check body has type τ_2)

(app) $$\frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau}$$

When there are no assumptions, we write $\vdash e : \tau$.

Theorem 1 (Type Soundness). *If $\vdash e : \tau$ and $e \to^* e'$, then e' is not stuck due to a type error.*

Our proof will be broken into a number of pieces. First, there are some lemmas to get out of the way:

Theorem 2 (Substitution). *If $\{x \mapsto \tau'\} \vdash e_1 : \tau$ and $\vdash e_2 : \tau'$, then $\vdash e_1[e_2/x] : \tau$.*

Proof idea. By induction on the height of the proof that $\{x \mapsto \tau'\} \vdash e_1 : \tau$. (Note that we only need to consider substituting values, but in a call-by-name setting, you'd have to consider general [closed] expressions so it's good to do the more general case here.) □

Theorem 3 (Canonical Forms). *If $\vdash v : \tau$ then*

1. *if $\tau = \mathsf{b}$, then $v = c$ for some constant c*
2. *if $\tau = \tau_1 \to \tau_2$, then $v = \lambda x.\, e$ for some function $\lambda x.\, e$*

Proof idea. By inspection of the typing rules. □

Theorem 4 (Preservation). *If $\vdash e : \tau$ and $e \to e'$, then $\vdash e' : \tau$.*

Proof idea. By induction on the height of the proof that $e \to e'$ (using Substitution as a lemma). □

Theorem 5 (Progress). *If $\vdash e : \tau$, then either $e = v$ for some value v, or else there exists an e' such that $e \to e'$ (using Canonical forms as a lemma.)*

Proof idea. By induction on the height of the proof that $\vdash e : \tau$. □

Finally, we can prove our Type Soundness theorem by induction on the length n of the sequence $e \to^n e'$. We are assuming that $\vdash e : \tau$, so when $n = 0$, $e' = e$ and thus, it is trivially the case that $\vdash e' : \tau$. By Progress, e' is either a value or else it can step, so it is not stuck.

Suppose the theorem holds up to n and we have $e \to^{n+1} e'$. Then we have $e \to^n e'' \to e'$. By our induction hypothesis, $\vdash e'' : \tau$. So by Preservation, $\vdash e' : \tau$. Then by Progress, e' is either a value or else it can take a step, so it is not stuck.

Type Checking and Type Inference

In class, we discussed Milner's algorithm W at a very high level. The algorithm looks something like this, where $\mathsf{TC}(\Gamma, e)$ is a function that takes a set of typing assumptions (Γ) and an expression to type-check (e) and returns a type τ, together with a set of equations S on types.

$$
\begin{array}{rcl}
\mathsf{TC}(\Gamma, c) & = & (\mathsf{b}, S) \\
\mathsf{TC}(\Gamma, x) & = & (\Gamma(x), S) \\
\mathsf{TC}(\Gamma, \lambda x.\, e) & = & \mathsf{let} \\
 & & \quad ? = \mathsf{fresh\_type\_variable}() \\
 & & \quad (t, s) = \mathsf{TC}(\Gamma[x \mapsto ?], e) \\
 & & \mathsf{in} \\
 & & \quad (? \to \tau, S) \\
 & & \mathsf{end} \\
\mathsf{TC}(\Gamma, e_1\ e_2) & = & \mathsf{let} \\
 & & \quad (\tau_1, S_1) = \mathsf{TC}(\Gamma, e_1) \\
 & & \quad (\tau_2, S_2) = \mathsf{TC}(\Gamma, e_2) \\
 & & \quad ? = \mathsf{fresh\_type\_variable}() \\
 & & \quad S = S_1 \cup S_2 \cup \{\tau_1 = \tau 2 \to ?\} \\
 & & \mathsf{in} \\
 & & \quad (?, S) \\
 & & \mathsf{end}
\end{array}
$$

Suppose $\mathsf{TC}(\Gamma, e) = (\tau, S)$. And suppose that we solve for the unknown type variables in S (i.e., we discover that $?_1 = \mathsf{b}$, $?_2 = \mathsf{b} \to \mathsf{b}$, etc.) If we apply the solution to τ (i.e., compute $\tau[\mathsf{b}/?_1, (\mathsf{b} \to \mathsf{b})/?_2, \ldots]$, then that is the type of the expression. Of course, there may not be a solution to the equations in S. They could have nonsensical things like $\mathsf{b} = \mathsf{b} \to \mathsf{b}$ or $? = ? \to \mathsf{b}$. If this is the case, then there is a type-error in the program.

We can solve for the unknowns by rewriting the equations until they are simplified:

$$S \cup \{\mathsf{b} = \mathsf{b}\} \to S$$

$$S \cup \{\tau_1 \to \tau_2 = \tau_1' \to \tau_2'\} \to S \cup \{\tau_1 = \tau_1', \tau_2 = \tau_2'\}$$

$$\frac{? \notin \tau \qquad S[\tau/?] \to S'}{S \cup \{? = \tau\} \to S' \cup \{? = \tau\}}$$

If we keep simplifying the equations, we'll either get into a situation where there is some unsatisfiable equation, such as:

$$\mathsf{b} = \tau_1 \to \tau_2$$

or

$$? = ? \rightarrow \tau$$

or else we'll end up with equations of the form:

$$?_1 = \tau_1, ?_2 = \tau_2, \ldots, ?_n = \tau_n$$

Then, the final answer is obtained by taking the result type of the expression and substituting the types for the unknown type variables:

$$\tau[\tau_1/?_1, \ldots, \tau_n/?_n]$$

If we get stuck in the simplification process, then there was a type-error somewhere. That is, there is no way to prove that the expression is well-typed.

In practice, the process of solving the equations is done online with a fast algorithm called resolution or unification (due to Robinson in 1968). The algorithm runs in (almost) linear time which makes it very fast and very scalable. It's implemented by unknowns (?) as reference cells that are initially null. When we encounter an equation of the form $? = \tau$, we simply set the pointer in ? to point to τ. When comparing two types, we always go through the ? pointers (if present). And to get the (almost) linear time algorithm, we must do path compression in the form of some sort of union/find data structure.

Solving the equations online tends to produce better error messages since we detect inconsistencies earlier. But still, you can get some strange error messages because the equation simplification process (unification) it to write the recursive factorial function. In ML, we'd write:

```
fun fact(n) = if (n <= 1) then 1 else n * fact(n-1)
```

Note that, again, this is an equation, not really a definition so we want to solve the equation for the real fact function.

One way to do this is to define:

```
val fact_body = fn f => fn n => if (n <= 1) then 1 else n * fact(n-1)
```

Then we can define:

```
val fact = fix(fact_body)
```

Note that

```
fact = fix(fact_body) =>
       fact_body (fix(fact_body)) =>
       fn n => if (n <= 1) then 1 else n * (fix(fact_body))(n-1) =
       fn n => if (n <= 1) then 1 else n * fact(n-1)
```

So, if you apply fact to a number (e.g., 3) here's what happens:

```
fact(3) = (fn n => if (n <= 1) then 1 else n * fact(n-1))(3) =>
          if (3 <= 1) then 1 else 3 * (fix(fact_body))(3-1) =>
          3 * (fix(fact_body))(2) =>
          3 * (fn n => if (n <= 1) then 1 else n * fact(n-1))(2) =>
```

```
3 * (if (2 <= 1) then 1 else 2 * (fix(fact_body))(2-1)) =>
3 * 2 * (fix(fact_body))(1) =>
3 * 2 * (fn n => if (n <= 1) then 1 else n * fact(n-1))(1) =>
3 * 2 * (if (1 <= 1) then 1 else n * fact(n-1)) =>
3 * 2 * 1 =>
6
```

What is the type of fact? In a call-by-value setting it needs to be something like:

$$\frac{\Gamma \vdash e : (\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)}{\Gamma \vdash \mathsf{fix}(e) : \tau_1 \to \tau_2}$$

[NB: I had a typo in the original notes here. The above rule is now right.]

So, we pass to fix a function which abstracts the recursive call as an extra argument (the first $\tau_1 \to \tau_2$). The function then takes in a τ_1 value and returns a τ_2 (typically by recursing.) If we pass such a function to fix, then it goes ahead and unrolls the loop for us one time, giving us back a function which takes a τ_1 to a τ_2. Every time we call the function, it gets unrolled one more time. This is exactly the same thing as what happend in the semantics of while-loops for the denotational semantics of IMP.

Homework

1. Prove the Substitution lemma. Include your proof in an ML comment at the beginning of your code for problem 2.
2. Write a type-checker for the simply-typed lambda calculus with unit, pairs, and sums. Use the following definitions:

```
datatype Type = Int_t | Arrow_t of Type*Type | Unit_t | Prod_t of Type*Type |
                Sum_e*Type

type var = string
datatype exp =
   Var of string |        (* x *)
   Int of int |           (* i *)
   Plus of exp*exp |      (* e1 + e2 *)
   Fn of var*Type*exp |   (* \x:t.e *)
   App of exp*exp |       (* e1(e2) *)
   Unit |                 (* () *)
   Pair of exp*exp |      (* (e1,e2) *)
   Num1 of exp |          (* #1 e *)
   Num2 of exp |          (* #2 e *)
   Inl of Type * Type * exp |    (* Inl[t1+t2](e) *)
   Inr of Type * Type * exp |    (* Inr[t1+t2](e) *)
   Case of exp * (var * exp) * (var * exp)  (* case e of Inl(x) => e1
                                                       | Inr(y) => e2 *)
exception TypeError of string
type assump = var -> Type   (* typing assumptions *)
val empty_assump : assump = fn x => raise TypeError("unbound variable")
```

   Your job is to write the function:

```
val type_check : assump * exp -> Type
```