CS 7400 October 21-26, 2009
Dr. Wand Readings: EOPL3, Chapter 7

Lecture 6: Types

Key Concepts:

type analysis
type error
sound analysis, conservative analysis
value v is of type t, value v has type t
expression e is assigned type t
type checking
type environment
type inference
type variable
type equation
substitution
unification
no-occurrence invariant
occurrence check

6.1 Overview

Goal: analyze a program to predict whether evaluation of a program is *safe*, that is whether the evaluation will proceed without certain kinds of errors.

For us: an evaluation is *safe* iff:

1. for every evaluation of a variable var, the variable is bound.
2. for every evaluation of a difference expression (`diff-exp` e_1 e_2), the values of e_1 and e_2 are both `num-vals`.
3. For every evaluation of an expression of the form (`zero?-exp` e_1), the value of e_1 is a `num-val`.
4. For every evaluation of a conditional expression (`if-exp` e_1 e_2 e_3), the value of e_1 is a `bool-val`.
5. for every evaluation of a procedure call (`proc-call` e_1 e_2), the value of e_1 is a `proc-val`.

We call violations of these conditions *type errors*.

Definition of what constitutes a type error may differ from language to language.

If we had multiple arguments, calling a procedure on the wrong number of arguments would be a type error.

A safe evaluation may:

- fail for other reasons: division by zero, taking the `car` of an empty list, etc. (too hard to analyze for these).
- run infinitely (too hard; undecidable in general).

Goal: write a procedure that looks at the program text and either accepts or rejects it.

- If the analysis accepts the program, then we can be sure evaluation of the program is safe. If the analysis cannot be sure that evaluation will be safe, it must reject the program. In this case, we say that the analysis is *sound* (or *conservative*).
- We'd like the analysis to accept enough programs to be useful.

Examples:

`if 3 then 88 else 99`	reject: non-boolean test
`proc (x) (3 x)`	reject: non-proc-val rator
`proc (x) (x 3)`	accept
`proc (f) proc (x) (f x)`	accept
`let x = 4 in (x 3)`	reject: non-proc-val rator
`(proc (x) (x 3)` `4)`	reject: same as preceding example
`let x = iszero?(0)` `in -(3, x)`	reject: non-integer argument to a diff-exp
`(proc (x) -(3,x)` `iszero?(0))`	reject: same as preceding example
`let f = 3` `in proc (x) (f x)`	reject: non-proc-val rator
`(proc (f) proc (x) (f x)` `3)`	reject: same as preceding example
`letrec f(x) = (f -(x,-1))` `in (f 1)`	accept: non-terminating but safe

6.2 Values and their types

Since the safety conditions only talk about `num-val`, `bool-val`, and `proc-val`, one might think that it would be enough to keep track of these three types. But that is not enough: if all we know is that `f` is bound to a `proc-val`, then we cannot draw any conclusions whatsoever about the value of `(f 1)`. From this argument, we learn that we need to keep track of finer information about procedures. This finer information is called the *type structure* of the language.

Our languages will have a very simple type structure. For the moment, consider the expressed values of LETREC. These values include only 1-argument procedures, but dealing with multi-argument procedures requires some additional work but does not require any new ideas.

Grammar for Types

Type ::= `int`
`int-type ()`

Type ::= `bool`
`bool-type ()`

Type ::= `(`*Type* `->` *Type*`)`
`proc-type (arg-type result-type)`

The value of `3` has type `int`.

The value of `-(33,22)` has type `int`.

The value of `zero?(11)` has type `bool`.

The value of `proc (x) -(x,11)` has type `(int -> int)`.
When given an integer, it returns an integer.

The value of `proc (x) let y = -(x,11) in -(x,y)`
has type `(int -> int)`.
When given an integer, it returns an integer.

The value of `proc (x) if x then 11 else 22`
has type `(bool -> int)`.
When given a boolean, it returns an integer.

The value of `proc (x) if x then 11 else zero?(11)` has no type in our type system.
When given a boolean it might return either an integer or a boolean, and we have no type that describes this behavior.

The value of `proc (x) proc (y) if y then x else 11`
has type `(int -> (bool -> int))`.
When given a boolean, it returns a procedure from booleans to integers.

The value of `proc (f) if (f 3) then 11 else 22`
has type `((int -> bool) -> int)`.
When given a procedure from integers to booleans, it returns an integer.

The value of `proc (f) (f 3)`
has type `((int -> `t`) -> `t`)` for any type t.
When given a procedure of type `(int -> `t`)`, it returns a value of type t.

The value of `proc(f) proc (x) (f (f x))`
has type `((`t` -> `t`) -> (`t` -> `t`))` for any type t.
When given a procedure of type `(`t` -> `t`)`, it returns another procedure which, when given an argument of type t, returns a value of type t.

Let's write down a definition that captures these examples. It will be defined by induction on t. (See, we are following the Design Recipe!)

Definition 1 *The property of an expressed value v being of type t is defined by induction on t:*

- *An expressed value is of type* `int` *iff it is a* `num-val`.
- *It is of type* `bool` *iff it is a* `bool-val`.
- *It is of type* $(t_1 \to t_2)$ *iff it is a* `proc-val` *with the property that if it is given an argument of type t_1, then one of the following things happen:*
  1. *it returns a value of type t_2*
  2. *it fails to terminate*
  3. *it fails with an error other than a type error.*

We occasionally say "v has type t" instead of "v is of type t."

Puzzle: in this system can a value val have more than one type?

Puzzle: For the language LETREC, is it decidable whether an expression e has a value that is of type t?

6.3 Assigning a type to an expression

Requirement: Write a procedure `type-of` which, given an expression (call it *exp*) and a *type environment* (call it *tenv*) mapping each variable to a type, assigns to *exp* a type t with the property that:

Specification of `type-of`

Whenever *exp* is evaluated in an environment in which each variable has a value of the type specified for it by *tenv*, one of the following happens:

- the resulting value has type t,
- the evaluation does not terminate, or
- the evaluation fails on an error other than a type error.

Another way of writing the permissible outcomes:

The evaluation does not cause a type error, and if it terminates, its value is of type t.

Our analysis will be based on the principle that if we can predict the types of the values of each of the subexpressions in an expression, we can predict the type of the value of the expression.

We'll use this idea to write a specification for `type-of`. We will write this specification as a set of inference rules, as we have done elsewhere. Assume that *tenv* is a *type environment* mapping each variable to its type. Then we should have:

Simple typing rules

$$\textit{tenv} \vdash \texttt{(const-exp } \textit{num}\texttt{)} : \texttt{int}$$

$$\textit{tenv} \vdash \texttt{(var-exp } var\texttt{)} : \textit{tenv}(var)$$

$$\frac{\textit{tenv} \vdash e_1 : \texttt{int}}{\textit{tenv} \vdash \texttt{(zero?-exp } e_1\texttt{)} : \texttt{bool}}$$

$$\frac{\textit{tenv} \vdash e_1 : t_1 \qquad [var = t_1]\textit{tenv} \vdash \textit{body} : t_2}{\textit{tenv} \vdash \texttt{(let-exp } var\ e_1\ \textit{body}\texttt{)} : t_2}$$

$$\frac{\begin{array}{l}\textit{tenv} \vdash e_1 : \texttt{bool} \\ \textit{tenv} \vdash e_2 : t \\ \textit{tenv} \vdash e_3 : t\end{array}}{\textit{tenv} \vdash \texttt{(if-exp } e_1\ e_2\ e_3\texttt{)} : t}$$

$$\frac{\textit{tenv} \vdash \textit{rator} : (t_1 \to t_2) \qquad \textit{tenv} \vdash \textit{rand} : t_1}{\textit{tenv} \vdash \texttt{(call-exp } \textit{rator}\ \textit{rand}\texttt{)} : t_2}$$

What about procedures?

If `proc` (x) e has type $(t_1 \rightarrow t_2)$, then it is intended to be called on an argument of type t_1. When its body e is evaluated, the variable x will be bound to a value of type t_1.

This suggests the following rule:

$$\frac{[var = t_1]\textit{tenv} \vdash \textit{body} : t_2}{\textit{tenv} \vdash \texttt{(proc-exp}\ var\ \textit{body}\texttt{)} : (t_1 \rightarrow t_2)}$$

There's only one problem: if we are trying to compute the type of a `proc` expression, how are we going to find the type t_1 for the bound variable? It is nowhere to be found.

There are two standard designs for rectifying this situation:

- *Type Checking*: In this approach the programmer is required to supply the missing information about the types of bound variables, and the type-checker deduces the types of the other expressions and checks them for consistency.
- *Type Inference*: In this approach the type-checker attempts to *infer* the types for the bound variables based on how the variables are used in the program. If the language is carefully designed, the type-checker can infer all or most of the types of the bound variables.

We will study each of these in turn.

6.4 CHECKED: A Type-Checked Language

New language will be the same as LETREC, except that we require the programmer to include the types of all bound variables (except for `let`). For `letrec`-bound variables, we also require the programmer to specify the result type of the procedure as well.

```
proc (x : int) -(x,1)

letrec
  int double (x : int) = if zero?(x)
                         then 0
                         else -((double -(x,1)), -2)
in double

proc (f : (bool -> int)) proc (n : int) (f zero?(n))
```

The result type of `double` is `int`, but the type of `double` itself is `(int→int)`.

In Java or C one might write something more like

```
proc (int x) -(x,1)

letrec
  int double(int x) = if zero?(x) ...
in ...
```

This is confusing:

- it confuses the result type of `double` (which is `int`), with the type of `double` (which is `(int -> int)`).
- If the types are long, it's hard to read, as in the 3rd example above:

  ```
  proc ((bool -> int) f) proc (int n) (f zero?(n))
  ```

  This is not much of a factor in Java, but it's a problem in C, and we'll see even more complicated types when we get to modules.

Changed productions for CHECKED

Expression ::= `proc (`*Identifier* `:` *Type*`)` *Expression*

```
proc-exp (var ty body)
```

Expression ::= `letrec`
Type *Identifier* `(`*Identifier* `:` *Type*`) =` *Expression*
`in` *Expression*

```
letrec-exp
  (result-type proc-name bound-var bound-var-type
   proc-body
   letrec-body)
```

Rule for `proc-exps`:

$$\frac{[var = t_{var}]tenv \vdash body \ : \ t_{res}}{tenv \vdash (\texttt{proc-exp} \ var \ \ t_{var} \ \ body) \ : (t_{var} \rightarrow t_{res})}$$

What about `letrec`? A typical `letrec` looks like

$$\begin{array}{l}\texttt{letrec}\\ \quad t_{res}\ p\ \texttt{(}x : t_{var}\texttt{)}\ \texttt{=}\ e_{\textit{proc-body}}\\ \texttt{in}\ e_{\textit{letrec-body}}\end{array}$$

This expression declares a procedure named p, with formal parameter x of type t_{var} and body $e_{\textit{proc-body}}$. Hence the type of p should be $t_{var} \rightarrow t_{res}$.

Each of the expressions in the `letrec`, $e_{\textit{proc-body}}$ and $e_{\textit{letrec-body}}$, must be checked in a type environment where each variable is given its correct type. We can use our scoping rules to determine what variables are in scope, and hence what types should be associated with them.

In $e_{\textit{letrec-body}}$, the procedure name p is in scope. As suggested above, p is declared to have type $t_{var} \rightarrow t_{res}$. Hence $e_{\textit{letrec-body}}$ should be checked in the type environment

$$\textit{tenv}_{\textit{letrec-body}} = [p = (t_{var} \rightarrow t_{res})]\textit{tenv}$$

What about $e_{\textit{proc-body}}$? In $e_{\textit{proc-body}}$, the variable p is in scope, with type $t_{var} \rightarrow t_{res}$, and the variable x is in scope, with type t_{var}. Hence the type environment for $e_{\textit{proc-body}}$ should be

$$\textit{tenv}_{\textit{proc-body}} = [x = t_{var}]\textit{tenv}_{\textit{letrec-body}}$$

Furthermore, in this type environment, $e_{\textit{proc-body}}$ should have result type t_{res}.

Writing this down as a rule, we get:

$$\frac{\begin{array}{l}[var = t_{var}][p = (t_{var} \rightarrow t_{res})]\textit{tenv} \vdash e_{\textit{proc-body}} : t_{res}\\ [p = (t_{var} \rightarrow t_{res})]\textit{tenv} \vdash e_{\textit{letrec-body}} : t\end{array}}{\textit{tenv} \vdash \texttt{(letrec-exp}\ t_{res}\ p\ (var : t_{var})\ \texttt{=}\ e_{\textit{proc-body}}\ \ e_{\textit{letrec-body}}\texttt{)} : t}$$

6.4.1 Implementing the Checker

We will need to compare types for equality. We do this with the procedure `check-equal-type!`, compares two types and reports an error unless they are equal.

```
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (if (not (equal? ty1 ty2))
      (eopl:error 'check-equal-type!
        "Types didn't match: ~s != ~s in~%~s"
        (type-to-external-form ty1)
        (type-to-external-form ty2)
        exp))))
```

This uses `type-to-external-form`, which converts a type back into a list that is easy to read.

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type))))))
```

Now we can transcribe the rules into a program, just as we've been doing all along.

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))

(define type-of
  (lambda (exp tenv)
    (cases expression exp
```

$\boxed{\textit{tenv} \vdash \textit{num} : \texttt{int}}$

```
      (const-exp (num) (int-type))
```

$\boxed{\textit{tenv} \vdash var : \textit{tenv}(var)}$

```
      (var-exp (var) (apply-tenv tenv var))
```

$$\boxed{\frac{\textit{tenv} \vdash e_1 : \texttt{int} \qquad \textit{tenv} \vdash e_2 : \texttt{int}}{\textit{tenv} \vdash (\texttt{diff-exp}\ e_1\ e_2) : \texttt{int}}}$$

```
      (diff-exp (exp1 exp2)
        (let ((ty1 (type-of exp1 tenv))
              (ty2 (type-of exp2 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (check-equal-type! ty2 (int-type) exp2)
          (int-type)))
```

$$\boxed{\frac{\textit{tenv} \vdash e_1 : \texttt{int}}{\textit{tenv} \vdash (\texttt{zero?-exp}\ e_1) : \texttt{bool}}}$$

```
      (zero?-exp (exp1)
        (let ((ty1 (type-of exp1 tenv)))
          (check-equal-type! ty1 (int-type) exp1)
          (bool-type)))
```

$$\frac{\textit{tenv} \vdash e_1 : \texttt{bool} \quad \textit{tenv} \vdash e_2 : t \quad \textit{tenv} \vdash e_3 : t}{\textit{tenv} \vdash \texttt{(if-exp } e_1\ e_2\ e_3\texttt{)} : t}$$

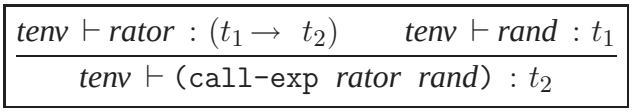
```
(if-exp (exp1 exp2 exp3)
  (let ((ty1 (type-of exp1 tenv))
        (ty2 (type-of exp2 tenv))
        (ty3 (type-of exp3 tenv)))
    (check-equal-type! ty1 (bool-type) exp1)
    (check-equal-type! ty2 ty3 exp)
    ty2))
```

$$\frac{\textit{tenv} \vdash e_1 : t_1 \qquad [var = t_1]\textit{tenv} \vdash \textit{body} : t_2}{\textit{tenv} \vdash \texttt{(let-exp } var\ e_1\ \textit{body}\texttt{)} : t_2}$$

```
(let-exp (var exp1 body)
  (let ((exp1-type (type-of exp1 tenv)))
    (type-of body
      (extend-tenv var exp1-type tenv))))
```

$$\frac{[var = t_{var}]\textit{tenv} \vdash \textit{body} : t_{res}}{\textit{tenv} \vdash \texttt{(proc-exp } var\ t_{var}\ \textit{body}\texttt{)} : (t_{var} \rightarrow\ t_{res})}$$

```
(proc-exp (var var-type body)
  (let ((result-type
         (type-of body
           (extend-tenv var var-type tenv))))
    (proc-type var-type result-type)))
```




```
(call-exp (rator rand)
  (let ((rator-type (type-of rator tenv))
        (rand-type  (type-of rand tenv)))
    (cases type rator-type
      (proc-type (arg-type result-type)
        (begin
          (check-equal-type! arg-type rand-type rand)
          result-type))
      (else
        (eopl:error 'type-of
          "Rator not a proc type:~%~s~%had rator type ~s"
          rator (type-to-external-form rator-type))))))
```

$$\frac{[var = t_{var}][p = (t_{var} \rightarrow\ t_{res})]\textit{tenv} \vdash e_{\textit{proc-body}} : t_{res} \quad [p = (t_{var} \rightarrow\ t_{res})]\textit{tenv} \vdash e_{\textit{letrec-body}} : t}{\textit{tenv} \vdash \texttt{(letrec-exp}\ t_{res}\ p\ (var : t_{var})\ \texttt{=}\ e_{\textit{proc-body}}\ \ e_{\textit{letrec-body}}\texttt{)} : t}$$

```
(letrec-exp (proc-result-type proc-name
             bound-var bound-var-type
             proc-body
             letrec-body)
  (let ((tenv-for-letrec-body
         (extend-tenv
           proc-name
           (proc-type
             bound-var-type proc-result-type)
           tenv)))
    (let ((proc-body-type
           (type-of proc-body
             (extend-tenv
               bound-var
               bound-var-type
               tenv-for-letrec-body))))
      (check-equal-type!
        proc-body-type proc-result-type proc-body)
      (type-of letrec-body tenv-for-letrec-body)))))))
```

6.5 INFERRED: a Language with Type Inference

Writing down the types in the program may be helpful for design and documentation, but it can be time-consuming. Another design is to have the compiler figure out the types of all the variables, based on observing how they are used, and utilizing any hints the programmer might give. Surprisingly, for a carefully-designed language, the compiler can *always* infer the types of the variables. This strategy is called *type inference*. We can do it for languages like LETREC, and it scales up to reasonably-sized languages.

For our case study in type inference, we start with the language of CHECKED. We then change the language so that all the types are optional. In place of a missing type expression, we use the marker `?`. Hence a typical program looks like

```
letrec
  ? foo (x : ?) = if zero?(x) then 1 else -(x, (foo -(x,1)))
in foo
```

Each question mark (except, of course, for the one in `zero?`) indicates a place where a type must be inferred.

Since types are optional, we may also fill in some of the `?`'s with types, as in

```
letrec
  ? even (x : int) = if zero?(x) then 1 else (odd -(x,1))
  bool odd (x : ?) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

To specify this syntax, we add a new non-terminal, *Optional-type*, and we modify the productions for `proc` and `letrec` to use optional types instead of types.

Optional-type ::= `?`

```
no-type ()
```

Optional-type ::= *Type*

```
a-type (ty)
```

Expression ::= `proc (`*Identifier* `:` *Optional-type*`)` *Expression*

```
proc-exp (var otype body)
```

Expression ::= `letrec`
Optional-type Identifier `(`*Identifier* `:` *Optional-type*`) =` *Expression*
`in` *Expression*

```
letrec-exp
  (proc-result-otype proc-name
   bound-var bound-var-otype proc-body
   letrec-body)
```

The omitted types will be treated as *unknowns* that we need to find. We do this by traversing the abstract syntax tree and generating equations between these types, possibly including these unknowns. We then solve the equations for the unknown types.

To see how this works, we need names for the unknown types. For each expression e or bound variable x, let t_e or t_x denote the type of the expression or bound variable.

For each node in the abstract syntax tree of the expression, the type rules dictate some equations that must hold between these types.

For our PROC language, the equations are:

$$
\begin{array}{lll}
\texttt{(diff-exp } e_1 \ e_2\texttt{)} & : & t_{e_1} = \texttt{int} \\
 & & t_{e_2} = \texttt{int} \\
 & & t_{\texttt{(diff-exp } e_1 \ e_2\texttt{)}} = \texttt{int} \\
 & & \\
\texttt{(zero?-exp } e_1\texttt{)} & : & t_{e_1} = \texttt{int} \\
 & & t_{\texttt{(zero?-exp } e_1\texttt{)}} = \texttt{bool} \\
 & & \\
\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)} & : & t_{e_1} = \texttt{bool} \\
 & & t_{e_2} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}} \\
 & & t_{e_3} = t_{\texttt{(if-exp } e_1 \ e_2 \ e_3\texttt{)}} \\
 & & \\
\texttt{(proc-exp } var \ \textit{body}\texttt{)} & : & t_{\texttt{(proc-exp } var \ \textit{body}\texttt{)}} = (t_{var} \rightarrow t_{\textit{body}}) \\
 & & \\
\texttt{(call-exp } \textit{rator} \ \textit{rand}\texttt{)} & : & t_{\textit{rator}} = (t_{\textit{rand}} \rightarrow t_{\texttt{(call-exp } \textit{rator} \ \textit{rand}\texttt{)}})
\end{array}
$$

- The first rule says that the arguments and the result of a `diff-exp` must all be of type `int`.
- The second rule says that the argument of a `zero?-exp` must be an `int`, and its result is a `bool`.
- The third rule says that in an `if` expression, the test must be of type `bool`, and that the types of the two alternatives must be the same as the type of the entire `if` expression.
- The fourth rule says that the type of a `proc` expression is that of a procedure whose argument type is given by the type of its bound variable, and whose result type is given by the type of its body.
- The fifth rule says that in a procedure call, the operator must have the type of a procedure that accepts arguments of the same type as that of the operand, and that produces results of the same type as that of the calling expression.

If we had multiargument procedures and abstractions, the equations for procedures and procedure calls would be

$$(\texttt{proc}\ (x_1\ \ldots\ x_n)\ e) \quad : \quad t_{\texttt{proc}\ (x_1\ \ldots\ x_n)\ e} = (t_{x_1} * \ldots * t_{x_n}) \rightarrow t_e$$

$$(\texttt{call-exp}\ e_0\ e_1\ \ldots\ e_n) \quad : \quad t_{e_0} = (t_{e_1} * \ldots * t_{e_n}) \rightarrow t_{(\texttt{call-exp}\ e_0\ e_1\ \ldots\ e_n)}$$

To infer the type of an expression, we'll introduce a type variable for every subexpression and every bound variable, generate the constraints for each subexpression, and then solve the resulting equations. To see how this works, we will infer the types of several sample expressions.

Let us start with the expression `proc(f)proc(x)-((f 3),(f x))`. We begin by making a table of all the bound variables and applications in this expression, and assigning a type variable to each one.

Expression	**Type Variable**
`f`	t_f
`x`	t_x
`proc(f)proc(x)-((f 3),(f x))`	t_0
`proc(x)-((f 3),(f x))`	t_1
`-((f 3),(f x))`	t_2
`(f 3)`	t_3
`(f x)`	t_4

Now, for each compound expression, we can write down a type equation according to the rules above.

Expression	**Equations**
`proc(f)proc(x)-((f 3),(f x))`	1. t_0 = $t_f \rightarrow t_1$
`proc(x)-((f 3),(f x))`	2. t_1 = $t_x \rightarrow t_2$
`-((f 3),(f x))`	3. t_3 = `int`
	4. t_4 = `int`
	5. t_2 = `int`
`(f 3)`	6. t_f = `int` $\rightarrow t_3$
`(f x)`	7. t_f = $t_x \rightarrow t_4$

- Equation 1 says that the entire expression produces a procedure that takes an argument of type t_f and produces a value of the same type as that of `proc(x)-((f 3),(f x))`.
- Equation 2 says that `proc(x)-((f 3),(f x))` produces a procedure that takes an argument of type t_x and produces a value of the same type as that of `-((f 3),(f x))`.
- Equations 3–5 say that the arguments and the result of the subtraction in `-((f 3),(f x))` are all integers.

- Equation 6 says that `f` expects an argument of type `int` and returns a value of the same type as that of `(f 3)`
- Similarly equation 7 says that `f` expects an argument of the same type as that of `x` and returns a value of the same type as that of `(f x)`.

We can fill in t_f, t_x, t_0, t_1, t_2, t_3, and t_4 in any way we like, so long as they satisfy the equations

$$
\begin{aligned}
t_0 &= t_f \to t_1 \\
t_1 &= t_x \to t_2 \\
t_3 &= \texttt{int} \\
t_4 &= \texttt{int} \\
t_2 &= \texttt{int} \\
t_f &= \texttt{int} \to t_3 \\
t_f &= t_x \to t_4
\end{aligned}
$$

Our goal is to find values for the variables that make all the equations true. We can express such a solution as a set of equations where the left-hand sides are all variables. We call such a set of equations a *substitution*. The variables that occur on the left-hand side of some equation in the substitution are said to be *bound* in the substitution.

We can solve such equations systematically. This process is called *unification*.

We separate the state of our calculation into the set of equations still to be solved and the substitution found so far. Initially, all of the equations are to be solved, and the substitution found is empty.

Equations	**Substitution**
$t_0 = t_f \to t_1$	
$t_1 = t_x \to t_2$	
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \to t_3$	
$t_f = t_x \to t_4$	

We consider each equation in turn. If the equation's left-hand side is a variable, we add it to the substitution.

Equations	**Substitution**
$t_1 = t_x \to t_2$	$t_0 = t_f \to t_1$
$t_3 = \texttt{int}$	
$t_4 = \texttt{int}$	
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \to t_3$	
$t_f = t_x \to t_4$	

However, doing this may change the substitution. For example, our next equation gives a value for t_1. We need to propagate that information into the value for t_0, which contains t_1 on its right-hand side. So we substitute the right-hand side for each occurrence of t_1 in the substitution. This gets us:

Equations	**Substitution**
$t_3 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_4 = \texttt{int}$	$t_1 = t_x \rightarrow t_2$
$t_2 = \texttt{int}$	
$t_f = \texttt{int} \rightarrow t_3$	
$t_f = t_x \rightarrow t_4$	

If the right-hand side were a variable, we'd switch the sides and do the same thing. We can continue in this manner for the next three equations.

Equations	**Substitution**
$t_4 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_2 = \texttt{int}$	$t_1 = t_x \rightarrow t_2$
$t_f = \texttt{int} \rightarrow t_3$	$t_3 = \texttt{int}$
$t_f = t_x \rightarrow t_4$	

Equations	**Substitution**
$t_2 = \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow t_2)$
$t_f = \texttt{int} \rightarrow t_3$	$t_1 = t_x \rightarrow t_2$
$t_f = t_x \rightarrow t_4$	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$

Equations	**Substitution**
$t_f = \texttt{int} \rightarrow t_3$	$t_0 = t_f \rightarrow (t_x \rightarrow \texttt{int})$
$t_f = t_x \rightarrow t_4$	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

Now, the next equation to be considered contains t_3, which is already bound to `int` in the substitution. So we substitute `int` for t_3 in the equation. We would do the same thing for any other type variables in the equation. We call this *applying* the substitution to the equation.

Equations	**Substitution**
$t_f = \texttt{int} \rightarrow \texttt{int}$	$t_0 = t_f \rightarrow (t_x \rightarrow \texttt{int})$
$t_f = t_x \rightarrow t_4$	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$

We move the resulting equation into the substitution and update the substitution as necessary; this time no updating takes place since t_f does not occur in the substitution.

Equations	**Substitution**
$t_f = t_x \rightarrow t_4$	$t_0 = (\texttt{int} \rightarrow \texttt{int}) \rightarrow (t_x \rightarrow \texttt{int})$
	$t_1 = t_x \rightarrow \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \rightarrow \texttt{int}$

The next equation, $t_f = t_x \rightarrow t_4$, contains t_f and t_4, which are bound in the substitution, so we apply the substitution to this equation. This gets

Equations	**Substitution**
$\texttt{int} \to \texttt{int} = t_x \to \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

If neither side of the equation is a variable, we can simplify, yielding two new equations.

Equations	**Substitution**
$\texttt{int} = t_x$	$t_0 = (\texttt{int} \to \texttt{int}) \to (t_x \to \texttt{int})$
$\texttt{int} = \texttt{int}$	$t_1 = t_x \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$

We can process these as usual: We switch the sides of the first equation, add it to the substitution, and update the substitution, as we did before.

Equations	**Substitution**
$\texttt{int} = \texttt{int}$	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

The final equation, `int = int`, is always true, so we can discard it.

Equations	**Substitution**
	$t_0 = (\texttt{int} \to \texttt{int}) \to (\texttt{int} \to \texttt{int})$
	$t_1 = \texttt{int} \to \texttt{int}$
	$t_3 = \texttt{int}$
	$t_4 = \texttt{int}$
	$t_2 = \texttt{int}$
	$t_f = \texttt{int} \to \texttt{int}$
	$t_x = \texttt{int}$

We have no more equations, so we are done. We conclude from this calculation that our original expression `proc(f)proc(x)-((f 3),(f x))` should be assigned the type

$$\texttt{(int} \to \texttt{int)} \to \texttt{(int} \to \texttt{int)}$$

This is reasonable: The first argument `f` must take an `int` argument because it is given `3` as an argument. It must produce an `int`, because its value is used as an argument to the subtraction operator. And `x` must be an `int`, because it is also supplied as an argument to `f`.

Let us consider another example: `proc(f)(f 11)`. Again, we start by assigning type variables:

Expression	**Type Variable**
`f`	t_f
`proc(f)(f 11)`	t_0
`(f 11)`	t_1

Next we write down the equations

Expression	**Equations**
`proc(f)(f 11)`	t_0 `=` $t_f \rightarrow t_1$
`(f 11)`	t_f `=` `int`$\rightarrow t_1$

And next we solve:

Equations	**Substitution**
$t_0 = t_f \rightarrow t_1$	
$t_f =$ `int`$\rightarrow t_1$	

Equations	**Substitution**
$t_f =$ `int`$\rightarrow t_1$	$t_0 = t_f \rightarrow t_1$

Equations	**Substitution**
	$t_0 = ($`int`$\rightarrow t_1) \rightarrow t_1$
	$t_f =$ `int`$\rightarrow t_1$

This means that we can assign `proc(f)(f 11)` the type `(int` $\rightarrow t_1$`)` $\rightarrow t_1$, for any choice of t_1. Again, this is reasonable: we can infer that `f` must be capable of taking an `int` argument, but we have no information about the result type of `f`, and indeed for any t_1, this code will work for any `f` that takes an `int` argument and returns a value of type t_1. We say it is *polymorphic* in t_1.

Let's try a third example. Consider `if x then -(x,1) else 0`. Again, let's assign type variables to each subexpression that is not a constant.

Expression	**Type Variable**
`x`	t_x
`if x then -(x,1) else 0`	t_0
`-(x,1)`	t_1

We then generate the equations

Expression	**Equations**
`if x then -(x,1) else 0`	t_x `=` `bool`
	t_1 `=` t_0
	`int` `=` t_0
`-(x,1)`	t_x `=` `int`
	t_1 `=` `int`

Processing these equations as we did before, we get

Equations	**Substitution**
$t_x =$ `bool`	
$t_1 = t_0$	
`int` $= t_0$	
$t_x =$ `int`	
$t_1 =$ `int`	

Equations	**Substitution**
$t_1 = t_0$	$t_x =$ `bool`
`int` $= t_0$	
$t_x =$ `int`	
$t_1 =$ `int`	

Equations	**Substitution**
$\texttt{int} = t_0$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	**Substitution**
$t_0 = \texttt{int}$	$t_x = \texttt{bool}$
$t_x = \texttt{int}$	$t_1 = t_0$
$t_1 = \texttt{int}$	

Equations	**Substitution**
$t_x = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = \texttt{int}$
	$t_0 = \texttt{int}$

Since t_x is already bound in the substitution, we apply the current substitution to the next equation, getting

Equations	**Substitution**
$\texttt{bool} = \texttt{int}$	$t_x = \texttt{bool}$
$t_1 = \texttt{int}$	$t_1 = \texttt{int}$
	$t_0 = \texttt{int}$

Oops! We have inferred from these equations that `bool = int`. So in any solution of these equations, `bool = int`. But `bool` and `int` cannot be equal. Therefore there is no solution to these equations. Therefore it is impossible to assign a type to this expression. This is reasonable, since the expression `if x then -(x,1) else 0` uses `x` as both a boolean and an integer, which is illegal in our type system.

Let us do one more example. Consider `proc(f)zero?((f f))`. We proceed as before.

Expression	**Type Variable**
`proc (f) zero?((f f))`	t_0
`f`	t_f
`zero?((f f))`	t_1
`(f f)`	t_2

Expression	**Equations**
`proc (f) zero?((f f))`	t_0 `=` $t_f \rightarrow t_1$
`zero?((f f))`	t_1 `=` `bool`
	t_2 `=` `int`
`(f f)`	t_f `=` $t_f \rightarrow t_2$

And we solve as usual:

Equations	**Substitution**
$t_0 = t_f \rightarrow t_1$	
$t_1 =$ `bool`	
$t_2 =$ `int`	
$t_f = t_f \rightarrow t_2$	

Equations	**Substitution**
$t_1 =$ `bool`	$t_0 = t_f \rightarrow t_1$
$t_2 =$ `int`	
$t_f = t_f \rightarrow t_2$	

Equations	**Substitution**
$t_2 =$ `int`	$t_0 = t_f \rightarrow$ `bool`
$t_f = t_f \rightarrow t_2$	$t_1 =$ `bool`

Equations	**Substitution**
$t_f = t_f \rightarrow t_2$	$t_0 = t_f \rightarrow \texttt{bool}$
	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$

Equations	**Substitution**
$t_f = t_f \rightarrow \texttt{int}$	$t_0 = t_f \rightarrow \texttt{bool}$
	$t_1 = \texttt{bool}$
	$t_2 = \texttt{int}$

Now we have a problem. We've now inferred that $t_f = t_f \rightarrow \mathit{Int}$. But there is no type with this property, because the right-hand side of this equation is always larger than the left: If the syntax tree for t_f contains k nodes, then the right hand side will always contain $k+2$ nodes.

So if we ever deduce an equation of the form $tv = t$ where the type variable tv occurs in the type t, we must again conclude that there is no solution to the original equations. This extra condition is called the *occurrence check*.

This condition also means that the substitutions we build will satisfy the following invariant:

The no-occurrence invariant

No variable bound in the substitution occurs in any of the right-hand sides of the substitution.

Our code for solving equations will depend critically on this invariant.

On-board exercise: Find the type for

```
(proc(f) proc(g) proc(x) if (g x) then (f g) else 27
 proc(h) (h 11))
```

6.5.1 Substitutions

We will build the implementation in a bottom-up fashion. We first consider substitutions.

We represent unknown types (sometimes called *type variables*) as an additional variant of the `type` datatype. We do this using the same technique that we to add lexical addresses to our SLLGEN grammars. We add to the grammar the additional production

Type ::= `%tvar-type` *Number*
`tvar-type (serial-number)`

We call these extended types *type expressions*. A basic operation on type expressions is substitution of a type for a type variable, defined by

apply-one-subst `: type * tvar * type -> type`
usage: `(apply-one-subst ty0 tvar ty1) returns the type obtained by substituting ty1 for every occurrence of tvar in ty0. This is sometimes written ty0[tvar=ty1]`

```
(define apply-one-subst
  (lambda (ty0 tvar ty1)
    (cases type ty0
      (proc-type (arg-type result-type)
        (proc-type
          (apply-one-subst arg-type tvar ty1)
          (apply-one-subst result-type tvar ty1)))
      (tvar-type (sn)
        (if (equal? ty0 tvar) ty1 ty0))
      (else ty0))))
```

This procedure deals with substituting for a single type variable. It doesn't deal with full-fledged substitutions like those we had in the preceding section.

A substitution is a list of equations between type variables and types. Equivalently, we can think of this list as a function from type variables to types. We say a type variable is *bound* in the substitution if and only if it occurs on the left-hand side of one of the equations in the substitution.

We represent a substitution as a list of (type variable, type) pairs. The basic observer for substitutions is `apply-subst-to-type`. This walks through the type, replacing each type variable by its binding in the substitution. If a variable is not bound in the substitution, then it is left unchanged. The implementation uses the auxiliiary procedure `apply-subst-to-tvar-type` to look up the type variable in the substitution. This procedure uses Scheme procedure `assoc`, which returns either the matching (type variable, type) pair or `#f` if the given type variable is not the car of any pair in the list.

apply-subst-to-tvar-type `: tvar-type * subst -> type`

```
(define apply-subst-to-tvar-type
  (lambda (tvar-type subst)
    (let ((tmp (assoc tvar-type subst)))
      (if tmp
        (cdr tmp)
        tvar-type))))
```

apply-subst-to-type `: type * subst -> type`

```
(define apply-subst-to-type
  (lambda (ty subst)
    (cases type ty
      (int-type () (int-type))
      (bool-type () (bool-type))
      (proc-type (t1 t2)
        (proc-type
          (apply-subst-to-type t1 subst)
          (apply-subst-to-type t2 subst)))
      (tvar-type (sn)
        (apply-subst-to-tvar-type ty subst)))))
```

The constructors for substitutions are `empty-subst` and `extend-subst`. `(empty-subst)` produces a representation of the empty substitution. `(extend-subst` σ tv t`)` takes the substitution σ and adds the equation $tv = t$ to it, as we did in the preceding section. We write $\sigma[tv = t]$ for the resulting substitution. This was a two-step operation: first we substituted t for tv in each of the right-hand sides of the equations in the substitution, and then we added the equation $tv = t$ to the list. Pictorially,

$$\begin{pmatrix} tv_1 = t_1 \\ \vdots \\ tv_n = t_n \end{pmatrix}[tv = t] \quad = \quad \begin{pmatrix} tv = t \\ tv_1 = t_1[tv = t] \\ \vdots \\ tv_n = t_n[tv = t] \end{pmatrix}$$

This definition has the property that for any type t,

$$(t\sigma)[tv = t'] = t(\sigma[tv = t'])$$

This is a kind of associativity property.

The implementation of `extend-subst` follows the picture above. It substitutes t_0 for tv_0 in all of the existing bindings in σ_0, and then adds the binding for t_0.

```
(define empty-subst (lambda () '()))
```

extend-subst `: subst * tvar * type -> subst`
usage: `tvar not already bound in subst.`

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons
      (cons tvar ty)
      (map
        (lambda (p)
          (let ((tvar-type1 (car p))
                (type1 (cdr p)))
            (cons
              tvar-type1
              (apply-one-subst type1 tvar ty))))
        subst))))
```

This implementation does not depend on, nor does it attempt to enforce, the no-occurrence invariant. That is the job of the unifier, in the next section.

6.5.2 The Unifier

The main procedure of the unifier is `unifier`. The unifier performs one step of the inference procedure outlined above: It takes two types, `ty1` and `ty2`, a substitution `subst` that satisfies the no-occurrence invariant, and an expression `exp`. It returns the substitution that results from adding `ty1 = ty2` to `subst`. This substitution will also satisfy the no-occurrence invariant. If adding `ty1 = ty2` yields an inconsistency, then it reports an error, and blames the expression `exp`. This is typically the expression that gave rise to the equation `ty1 = ty2`.

Think of the substitution as a *store*, and an unknown type is a *reference* into that store. `unifier` produces the new store that is obtained by adding `ty1 = ty2` to the store.

This is an algorithm for which `cases` gives awkward code, so we use simple predicates and extractors on types instead.

unifier : type * type * subst * exp -> subst
usage: finds the smallest extension of subst
that unifies ty1[subst] and ty2[subst].
Raises an error if there is no such unifier.

```
(define unifier
  (lambda (ty1 ty2 subst exp)
    (let ((ty1 (apply-subst-to-type ty1 subst))
          (ty2 (apply-subst-to-type ty2 subst)))
      (cond
        ((equal? ty1 ty2) subst)
        ((tvar-type? ty1)
         (if (no-occurrence? ty1 ty2)
           (extend-subst subst ty1 ty2)
           (raise-occurrence-check! ty1 ty2 exp)))
        ((tvar-type? ty2)
         (if (no-occurrence? ty2 ty1)
           (extend-subst subst ty2 ty1)
           (raise-occurrence-check! ty2 ty1 exp)))
        ((and (proc-type? ty1) (proc-type? ty2))
         (let ((subst (unifier
                       (proc-type->arg-type ty1)
                       (proc-type->arg-type ty2)
                       subst exp)))
           (let ((subst (unifier
                         (proc-type->result-type ty1)
                         (proc-type->result-type ty2)
                         subst exp)))
             subst)))
        (else (raise-type-error! ty1 ty2 exp))))))
```

- First, as we did above, we apply the substitution to each of the types `ty1` and `ty2`.
- If the resulting types are the same, we return immediately. This corresponds to the step of deleting a trivial equation above.
- If `ty1` is an unknown type, then the no-occurrence invariant tells us that it is not bound in the substitution. Hence it must be unbound, so we propose to add `ty1 = ty2` to the substitution. But we need to perform the occurrence check, so that the no-occurrence invariant is preserved.
- If `ty2` is an unknown type, we do the same thing, reversing the roles of `ty1` and `ty2`.
- If neither `ty1` nor `ty2` is a type variable, then we can analyze further.

  If they are both `proc` types, then we simplify by equating the argument types, and then equating the result types in the resulting substitution.

  Otherwise, either one of `ty1` and `ty2` is `int` and the other is `bool`, or one is a `proc` type and the other is `int` or `bool`. In any of these cases, there is no solution to the equation, so an error is reported.

Last, we must implement the occurrence check. This is a straightforward recursion on the type

no-occurrence? `: tvar * type -> bool`
usage: `Returns #t iff is there no occurrence of tvar in ty.`

```
(define no-occurrence?
  (lambda (tvar ty)
    (cases type ty
      (proc-type (arg-type result-type)
        (and
          (no-occurrence? tvar arg-type)
          (no-occurrence? tvar result-type)))
      (tvar-type (serial-number) (not (equal? tvar ty)))
      (else #t))))
```

6.5.3 Building the Type Inferencer

We convert optional types to types with unknowns by creating a fresh type variable for each ?, using `otype->type`.

```
(define otype->type
  (lambda (otype)
    (cases optional-type otype
      (no-type () (fresh-tvar-type))
      (a-type (ty) ty))))

(define fresh-tvar-type
  (let ((sn 0))
    (lambda ()
      (set! sn (+ sn 1))
      (tvar-type sn))))
```

When we convert to external form, we represent a type variable by a symbol containing its serial number.

```
(define type-to-external-form
  (lambda (ty)
    (cases type ty
      (int-type () 'int)
      (bool-type () 'bool)
      (proc-type (arg-type result-type)
        (list
          (type-to-external-form arg-type)
          '->
          (type-to-external-form result-type)))
      (tvar-type (serial-number)
        (string->symbol
          (string-append
            "ty"
            (number->string serial-number)))))))
```

Now we can write `type-of`. It takes an expression, a type environment mapping program variables to type expressions, and a substitution satisfying the no-occurrence invariant, and it returns a type and a new no-occurrence substitution.

The type environment associates a type expression with each program variable. The substitution explains the meaning of each type variable in the type expressions. It is useful to think of the substitution as a *store*, and a type variable as *reference* into that store. Therefore, `type-of` returns two values: a type expression, and a substitution in which to interpret the type variables in that expression.

We implement this by creating a new datatype that contains the two values, and using that as the return value.

For each kind of expression, we recur on the subexpressions, passing along the solution so far in the substitution argument. Then we generate the equations for the current expression, according to the specification, and record these in the substitution by calling `unifier`.

```
;; answer = type * subst

(define-datatype answer answer?
  (an-answer
    (ty type?)
    (subst substitution?)))
```

type-of-program `: program -> type`

```
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1)
        (cases answer (type-of exp1
                               (init-tenv) (empty-subst))
          (an-answer (ty subst)
            (apply-subst-to-type ty subst)))))))
```

type-of : exp * tenv * subst -> answer

```
(define type-of
  (lambda (exp tenv subst)
    (cases expression exp

      (const-exp (num) (an-answer (int-type) subst))
```

(zero?-exp e_1)	:	$t_{e_1} = \texttt{int}$ $t_{\texttt{(zero?-exp } e_1\texttt{)}} = \texttt{bool}$

```
      (zero?-exp (exp1)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst2
                    (unifier ty1 (int-type) subst1 exp)))
              (an-answer (bool-type) subst2)))))
```

(diff-exp e_1 e_2)	:	$t_{e_1} = \texttt{int}$ $t_{e_2} = \texttt{int}$ $t_{\texttt{(diff-exp } e_1\ e_2\texttt{)}} = \texttt{int}$

```
      (diff-exp (exp1 exp2)
        (cases answer (type-of exp1 tenv subst)
          (an-answer (ty1 subst1)
            (let ((subst1
                    (unifier ty1 (int-type) subst1 exp1)))
              (cases answer (type-of exp2 tenv subst1)
                (an-answer (ty2 subst2)
                  (let ((subst2
                          (unifier ty2 (int-type)
                            subst2 exp2)))
                    (an-answer (int-type) subst2)))))))))
```

(if-exp e_1 e_2 e_3)	:	$t_{e_1} = \texttt{bool}$ $t_{e_2} = t_{\texttt{(if-exp } e_1\ e_2\ e_3\texttt{)}}$ $t_{e_3} = t_{\texttt{(if-exp } e_1\ e_2\ e_3\texttt{)}}$

```
(if-exp (exp1 exp2 exp3)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (ty1 subst)
      (let ((subst
              (unifier ty1 (bool-type) subst exp1)))
        (cases answer (type-of exp2 tenv subst)
          (an-answer (ty2 subst)
            (cases answer (type-of exp3 tenv subst)
              (an-answer (ty3 subst)
                (let ((subst
                        (unifier ty2 ty3 subst exp)))
                  (an-answer ty2 subst)))))))))))

(var-exp (var)
  (an-answer (apply-tenv tenv var) subst))

(let-exp (var exp1 body)
  (cases answer (type-of exp1 tenv subst)
    (an-answer (exp1-type subst)
      (type-of body
        (extend-tenv var exp1-type tenv)
        subst))))
```

(proc-exp *var otype body*) : $t_{\texttt{(proc-exp } var \texttt{ } body\texttt{)}} = (t_{var} \rightarrow t_{body})$

```
(proc-exp (var otype body)
  (let ((var-type (otype->type otype)))
    (cases answer (type-of body
                    (extend-tenv var var-type tenv)
                    subst)
      (an-answer (result-type subst)
        (an-answer
          (proc-type var-type result-type)
          subst)))))
```

(call-exp *rator rand*) : $t_{rator} = (t_{rand} \rightarrow t_{\texttt{(call-exp } rator \texttt{ } rand\texttt{)}})$

```
(call-exp (rator rand)
  (let ((result-type (fresh-tvar-type)))
    (cases answer (type-of rator tenv subst)
      (an-answer (rator-type subst)
        (cases answer (type-of rand tenv subst)
          (an-answer (rand-type subst)
            (let ((subst
                    (unifier
                      rator-type
                      (proc-type
                        rand-type result-type)
                      subst
                      exp)))
              (an-answer result-type subst)))))))))
```

(letrec-exp $t_{proc-result}$ *p* $(x : t_{var})$ = $e_{proc\text{-}body}$ $e_{letrec\text{-}body}$) :
$t_p = t_x \to t_{e_{proc\text{-}body}}$
$t_{e_{letrec\text{-}body}} = t_{(\texttt{letrec-exp}\ t_{proc-result}\ p\ (x : t_{var}) = e_{proc\text{-}body}\ e_{letrec\text{-}body})}$

```
(letrec-exp (proc-result-otype proc-name
             bound-var bound-var-otype
             proc-body
             letrec-body)
  (let ((proc-result-type
         (otype->type proc-result-otype))
        (proc-var-type
         (otype->type bound-var-otype)))
    (let ((tenv-for-letrec-body
           (extend-tenv
            proc-name
            (proc-type
              proc-var-type proc-result-type)
            tenv)))
      (cases answer (type-of proc-body
                     (extend-tenv
                       bound-var proc-var-type
                       tenv-for-letrec-body)
                     subst)
        (an-answer (proc-body-type subst)
          (let ((subst
                  (unifier
                    proc-body-type
                    proc-result-type
                    subst
                    proc-body)))
            (type-of letrec-body
              tenv-for-letrec-body
              subst)))))))

)))
```

Testing the inferencer is somewhat more subtle than testing our previous interpreters, because of the possibility of polymorphism. For example, if the inferencer is given `proc (x) x`, it might generate any of the external forms `(ty1 -> ty1)` or `(ty2 -> ty2)` or `(ty3 -> ty3)`, and so on. These may be different every time through the inferencer, so we won't be able to anticipate them when we write our test items. So when we compare the produced type to the correct type, we'll fail. We need to accept all of the alternatives above, but reject `(ty3 -> ty4)` or `(int -> ty17)`.

The inferencer produces its output in external form, so our solution must use that representation. To compare two types in external form, we standardize the names of the unknown types, by walking through each external form, renumbering all the type variables so that they are numbered starting with `ty1`. We can then compare the renumbered types with `equal?`.

To systematically rename each unknown type, we construct a substitution with `canonical-subst`. This is a straightforward recursion, with `table` playing the role of an accumulator. The length of `table` tells us how many distinct unknown types we have found, so we can use its length to give the number of the "next" `ty` symbol. This is similar to the way we used `length` in our "world's dumbest" store model.

Details are in the book.

6.6 Adding Polymorphism

Our type inference system is neat, but we'd like to be able to define *polymorphic* procedures: procedures with more than one type. For example, consider:

```
let id = proc (x) x
in if (id true) then (id 3) else (id 4)
```

This uses `id` at two types: `(bool -> bool)` and `(int -> int)`. We say that `id` is *polymorphic*.

Our system right now would reject this program. We would prefer it to infer a type like

$$(\forall t)(t \to t)$$

for `id`, and then use it for two different values of t.

We often use procedures like this. Consider a language with lists. Then we might define list types by:

A value is of type `list` t iff it is a list and all of its elements are of type t.

Then `cons` is of type
`(all t)(t * list t -> list t)`,
and `map` is of type
`(all s,t)((s -> t) * list s -> list t)`.

If there are only a few such procedures, we can special-case them in the type-checker, much as we did for `if` (which could be thought of as having the type `(all t)(bool * t * t -> t)`.

But of course we'd like to be able to define such procedures in the language, as in the example above.

To do this requires 4 steps:

1. We distinguish between types (which we've studied already), and *type schemes* or *quantified types*. A type scheme consists of a type and a set of *local* or *generic* or *instantiable* type variables.

2. We distinguish the "denoted" and "expressed" values in the type checker. The "denoted values" will always be type schemes– that is, the type environment will map identifiers to type *schemes*. But the types of expressions will still be types.

3. Whenever we find the type of an identifier, we create a new type, with fresh type variables substituted for each of the local type variables. For example, if `id` is associated with the type scheme `(all s)(s -> s)`, each time we try to do `(type-of-expression id tenv)`, we'll get a new type:

```
(tvar26 -> tvar26)
(tvar28 -> tvar28)
(tvar29 -> tvar29)
```

etc. This way we can use `id` several times in its scope at several different types, as desired.

This is like the automatic dereferencing in the implicit-store-operations design!

4. How do we put type schemes into the type environment?

In the conventional design, polymorphic values are declared only by `let`. Variables bound by `proc` or `letrec` are *monomorphic*: they will always have a type scheme with no generic type variables.

Variables bound by `let` should be as polymorphic as possible. If we have a `let`-bound variable, how do we figure out its generic type variables?

Let's consider some examples. The most important uses of polymorphic procedures are on pairs and lists. Since I don't want to change the type system, I'm going to show you how to code pairs as procedures.

```
let pair = proc(?x,?y) proc(?k) (k x y)
    fst = proc(?p) (p proc(?x,?y)x)
    snd = proc(?p) (p proc(?x,?y)y)
in (fst (pair 11 true))
```

Notice that if x and y are both terms that terminate without side-effects, then

$$\texttt{(fst (pair } x\ y\texttt{))} = x$$
$$\texttt{(snd (pair } x\ y\texttt{))} = y$$

so this works like a pair. `pair` has type `(t1 * t2 -> ((t1 * t2 -> t3) -> t3))`.

In INFERRED, this program works, and has type `int`. Same thing for

```
let pair = proc(?x,?y) proc(?k) (k x y)
    fst = proc(?p) (p proc(?x,?y)x)
in let p1 = (pair 11 true)
       p2 = (pair 12 true)
   in +((fst p1), (fst p2))
```

because the two occurrences of `pair` are used at the same type. But if we write

```
let pair = proc(?x,?y) proc(?k) (k x y)
    fst = proc(?p) (p proc(?x,?y)x)
    snd = proc(?p) (p proc(?x,?y)y)
in let p1 = (pair 11 true)
       p2 = (pair true 12)
   in +((fst p1), (snd p2))
```

then INFERRED rejects the program, because the two instances of `pair` are used at different types: there is only one `t1`, and it can't be both `int` and `bool`!

Similarly,

```
let pair = proc(?x,?y) proc(?k) (k x y)
    fst = proc(?p) (p proc(?x,?y)x)
    snd = proc(?p) (p proc(?x,?y)y)
in let p1 = (pair 11 true)
   in if snd(p1) then fst(p1) else 10
```

fails because the "target type" `t3` can't be both `int` and `bool`.

Both these programs should work with polymorphic `let`. For `let` x `=` e `in` e', we'd expect the generic type variables for x to be all the unbound type variables that show up in the type of e. `pair` gets assigned the type scheme

```
(all s,t,u)(s * t -> ((s*t -> u) -> u)
```

This would explain all the examples so far.

[Exercise: what are the type schemes for `fst` and `snd`?]

The type of the returned pair is `((s*t -> u) -> u)`.

In our example, `p1` is `let`-bound, so the type scheme for `p1` is
`(all u)((int * bool -> u) -> u)`

This is necessary, because `p1` is used here with two different values for `u`.

This design is called *Hindley-Milner polymorphism*.

But things get more complicated if the `let` is inside some other bindings. For example, consider

```
let pair = ... fst = ... snd = ... % as before
proc (x)
  let pairx = proc(y) (pair x y)
  in ...
```

What type scheme should be associated with `pairx`?

We associate a type variable `tx` with `x` and type variable `ty` with `y`, so the type associated with `pairx` should be
`(ty -> ((tx * ty -> u) -> u))`

But which of these type variables should be counted as generic? We can't count the type variable `tx`: it will be the type of `x`, which is unchanging inside the scope.

To understand this, let's consider some different things that might appear in the body of this procedure:

`add1((fst (pairx true)))` : from this we can deduce that `tx` must be ie `int`.
`if (fst (pairx true)) then 11 else 22` : from this we can deduce that `tx` must be `bool`

If we allowed `tx` to vary, we could write

```
if (fst (pairx true)) then 11 else add1((fst (pairx true)))
```

If `tx` were generic, this would be well-typed, because here the two uses of `pairx` differ in their values for `tx`. But this is unsound, because `x` cannot be both an integer and a boolean: you could get an error at execution time.

We conclude that the type scheme associated with `pairx` should be
`(all ty, u)(ty -> ((tx * ty -> u) -> u))`

Here `tx` is a free type variable; it will get its value from the surrounding type environment.

In general, an unbound type variable must be kept free if it is mentioned in the type of any variable that is in scope, that is, if and only if it appears unbound somewhere in (the range of) *tenv*.

We can write this down in rules.

$$\frac{\textit{tenv}(x) = (\texttt{all}\ x_1, \ldots x_n)t \quad tv_1, \ldots, tv_n \text{ fresh}}{\texttt{(type-of-expression}\ \langle\langle x \rangle\rangle\ \textit{tenv}\texttt{)} = t[tv_1/x_1, \ldots, tv_n/x_n]}$$

$$\frac{\begin{array}{l}\texttt{(type-of-expression}\ \langle\langle e_1 \rangle\rangle\ \textit{tenv}\texttt{)} = t_1 \\ \texttt{(type-of-expression}\ \langle\langle e_2 \rangle\rangle\ \textit{tenv}\texttt{)} = t_2 \\ \ldots \\ \texttt{(type-of-expression}\ \langle\langle e_n \rangle\rangle\ \textit{tenv}\texttt{)} = t_n \\ V_i = \text{fvars}(t_i) - \text{fvars}(\text{rng}(\textit{tenv})) \\ s_i = (\texttt{all}\ V_i)t_i \\ \texttt{(type-of-expression}\ \langle\langle \textit{body} \rangle\rangle\ [x_1 = s_1, \ldots, x_n = s_n]\textit{tenv}\texttt{)} = t\end{array}}{\texttt{(type-of-expression}\ \langle\langle \texttt{let}\ x_1 = e_1 \ldots x_n = e_n \text{ in } \textit{body} \rangle\rangle\ \textit{tenv}\texttt{)} = t}$$

6.6.1 Pieces of an implementation

We start by defining a datatype for type schemes:

```
(define-datatype type-scheme type-scheme?
  (a-type-scheme
    (local-tvars (list-of tvar?))
    (type type?)))
```

We modify `apply-env` to instantiate each type scheme that it finds:

```
(define apply-tenv
  (lambda (tenv sym)
    (let apply-tenv ((tenv tenv) (sym sym))
      (cases type-environment tenv
        (empty-tenv-record ()
          (eopl:error 'apply-tenv
            "Variable ~s unbound in type environment" sym))
        (extended-tenv-record (syms vals tenv)
          (let ((pos (list-find-position sym syms)))
            (if (number? pos)
              (instantiate-type-scheme (list-ref vals pos))  ; new!
              (apply-tenv tenv sym))))
        (typedef-record (name type tenv)
          (apply-tenv tenv sym))))))
```

Here, `instantiate-type-scheme` is to make a copy of the type with fresh tvars for each generic tvar.

This takes care of steps 1–3.

Next we need to deal with extensions of the type environment: how do type schemes get into the type environment?

Variables that are bound by `proc` or `letrec` get associated with monomorphic types, that is type schemes with no bound variables. To do this, we modify `extend-tenv`:

```
(define extend-tenv
  (lambda (syms types tenv)
    (extended-tenv-record
      syms
      (map (lambda (t) (a-type-scheme '() t)) types)
      tenv)))
```

We have to treat `let`-bound variables differently, so we redefine `type-of-let-exp`:

```
(define type-of-let-exp
  (lambda (ids rands body tenv)
    (let ((tenv-for-body
            (extend-tenv-with-type-schemes
              ids
              (types-of-expressions rands tenv)
              tenv)))
      (type-of-expression body tenv-for-body))))
```

where the procedure `extend-tenv-with-type-schemes` extends the the type environment with type schemes, following the the rules.

The rest is a matter of following the rules and implementing them.

Go out and have many things for dinner! Your stomach is polymorphic– it can digest many different things using the same algorithm!