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 \((\text{diff-exp } e_1 \ e_2)\), the values of \(e_1\) and \(e_2\) are both num-val.

3. For every evaluation of an expression of the form \((\text{zero?-exp } e_1)\), the value of \(e_1\) is a num-val.

4. For every evaluation of a conditional expression \((\text{if-exp } e_1 \ e_2 \ e_3)\), the value of \(e_1\) is a bool-val.

5. for every evaluation of a procedure call \((\text{proc-call } e_1 \ e_2)\), 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 \text{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:

```plaintext
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))            reject: same as preceding example
4)
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))   reject: same as preceding example
3)
letrec f(x) = (f -(x,-1)) accept: non-terminating but safe
in (f 1)
```
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**

\[
\begin{align*}
\text{Type} &::= \text{int} \\
&\quad \text{int-type ()} \\
\text{Type} &::= \text{bool} \\
&\quad \text{bool-type ()} \\
\text{Type} &::= (\text{Type} \rightarrow \text{Type}) \\
&\quad \text{proc-type (arg-type result-type)}
\end{align*}
\]

The value of 3 has type int.

The value of -(3,22) 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 \( \text{proc} \ (x) \ \text{let} \ y = -(x,11) \ \text{in} \ -(x,y) \)
has type \((\text{int} \to \text{int})\).
When given an integer, it returns an integer.

The value of \( \text{proc} \ (x) \ \text{if} \ x \ \text{then} \ 11 \ \text{else} \ 22 \)
has type \((\text{bool} \to \text{int})\).
When given a boolean, it returns an integer.

The value of \( \text{proc} \ (x) \ \text{if} \ x \ \text{then} \ 11 \ \text{else} \ \text{zero}?(11) \)
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 \( \text{proc} \ (x) \ \text{proc} \ (y) \ \text{if} \ y \ \text{then} \ x \ \text{else} \ 11 \)
has type \((\text{int} \to (\text{bool} \to \text{int}))\).
When given a boolean, it returns a procedure from booleans to integers.

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

The value of \( \text{proc} \ (f) \ (f \ 3) \)
has type \(((\text{int} \to t) \to t)\) for any type \(t\).
When given a procedure of type \((\text{int} \to t)\), it returns a value of type \(t\).

The value of \( \text{proc} \ (f) \ \text{proc} \ (x) \ \text{f} \ (f \ x) \)
has type \(((t \to t) \to (t \to t))\) for any type \(t\).
When given a procedure of type \((t \to t)\), 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 \( (t_1 \rightarrow t_2) \) 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 \( \text{val} \) 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 \texttt{type-of} which, given an expression (call it \textit{exp}) and a \textit{type environment} (call it \textit{tenv}) mapping each variable to a type, assigns to \textit{exp} a type \textit{t} with the property that:

Specification of \texttt{type-of}

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

- the resulting value has type \textit{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 \textit{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 \texttt{type-of}. We will write this specification as a set of inference rules, as we have done elsewhere. Assume that \textit{tenv} is a \textit{type environment} mapping each variable to its type. Then we should have:
Simple typing rules

\[
\begin{align*}
    \text{tenv} & \vdash (\text{const-exp } \text{num}) : \text{int} \\
    \text{tenv} & \vdash (\text{var-exp } \text{var}) : \text{tenv}(\text{var}) \\
    \text{tenv} & \vdash e_1 : \text{int} \\
    \hline
    \text{tenv} & \vdash (\text{zero?-exp } e_1) : \text{bool} \\
    \text{tenv} & \vdash e_1 : t_1 \quad [\text{var} = t_1] \quad \text{tenv} \vdash \text{body} : t_2 \\
    \hline
    \text{tenv} & \vdash (\text{let-exp } \text{var} \ e_1 \ \text{body}) : t_2 \\
    \text{tenv} & \vdash e_1 : \text{bool} \\
    \text{tenv} & \vdash e_2 : t \\
    \text{tenv} & \vdash e_3 : t \\
    \hline
    \text{tenv} & \vdash (\text{if-exp } e_1 \ e_2 \ e_3) : t \\
    \text{tenv} & \vdash \text{rator} : (t_1 \rightarrow t_2) \quad \text{tenv} \vdash \text{rand} : t_1 \\
    \hline
    \text{tenv} & \vdash (\text{call-exp } \text{rator} \ \text{rand}) : t_2
\end{align*}
\]
What about procedures?

If \( \text{proc} (x) \) \( e \) 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:

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

There’s only one problem: if we are trying to compute the type of a \( \text{proc} \) 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 \texttt{let}). For \texttt{letrec}-bound variables, we also require the programmer to specify the result type of the procedure as well.

\begin{verbatim}
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))
\end{verbatim}

The result type of \texttt{double} is \texttt{int}, but the type of \texttt{double} itself is \texttt{(int -> int)}. In Java or C one might write something more like

\begin{verbatim}
proc (int x) -(x,1)

letrec
  int double(int x) = if zero?(x) ... 
in ... 
\end{verbatim}

This is confusing:

\begin{itemize}
  \item it confuses the result type of \texttt{double} (which is \texttt{int}), with the type of \texttt{double} (which is \texttt{(int -> int)}).
  \item If the types are long, it’s hard to read, as in the 3rd example above:
\end{itemize}

\begin{verbatim}
proc ((bool -> int) f) proc (int n) (f zero?(n))
\end{verbatim}

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:

\[
\frac{[\text{var} = \text{t}_{\text{var}}]\text{tenv} \vdash \text{body} : \text{t}_{\text{res}}}{\text{tenv} \vdash (\text{proc-exp var t}_{\text{var}} \text{ body}) : (\text{t}_{\text{var}} \rightarrow \text{t}_{\text{res}})}
\]
What about \texttt{letrec}? A typical \texttt{letrec} looks like

\begin{verbatim}
letrec
t_{res} p \ (x : t_{var}) = e_{proc-body}
in e_{letrec-body}
\end{verbatim}

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

Each of the expressions in the \texttt{letrec}, \( e_{proc-body} \) and \( e_{letrec-body} \), 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 \( e_{letrec-body} \), the procedure name \( p \) is in scope. As suggested above, \( p \) is declared to have type \( t_{var} \rightarrow t_{res} \). Hence \( e_{letrec-body} \) should be checked in the type environment

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

What about \( e_{proc-body} \)? In \( e_{proc-body} \), 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 \( e_{proc-body} \) should be

\[ tenv_{proc-body} = [x = t_{var}]tenv_{letrec-body} \]

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

Writing this down as a rule, we get:

\[\begin{align*}
\forall var = t_{var} \forall p = (t_{var} \rightarrow t_{res})\ [tenv \vdash e_{proc-body} : t_{res}] \\
\forall p = (t_{var} \rightarrow t_{res})\ [tenv \vdash e_{letrec-body} : t]
\end{align*}\]

\[\vdash (letrec-exp \ t_{res} p \ (var : t_{var}) = e_{proc-body} \ e_{letrec-body}) : t \]
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
      (tenv ⊢ num : int
        (const-exp (num) (int-type)))
      (tenv ⊢ var : tenv(var)
        (var-exp (var) (apply-tenv tenv var)))
      (tenv ⊢ e1 : int  tenv ⊢ e2 : 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)))
      (tenv ⊢ e1 : int
        (zero?-exp (exp1)
          (let ((ty1 (type-of exp1 tenv)))
            (check-equal-type! ty1 (int-type) exp1)
            (bool-type)))
\[
\begin{align*}
\text{tenv} & \vdash e_1 : \text{bool} \\
\text{tenv} & \vdash e_2 : t \\
\text{tenv} & \vdash e_3 : t \\
\text{tenv} & \vdash (\text{if-exp } e_1 e_2 e_3) : t
\end{align*}
\]

(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))

\[
\begin{align*}
\text{tenv} & \vdash e_1 : t_1 \\
\text{tenv} & \vdash [\text{var} = t_1]\text{tenv} \vdash \text{body} : t_2 \\
\text{tenv} & \vdash (\text{let-exp } \text{var } e_1 \text{ body}) : t_2
\end{align*}
\]

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

\[
\begin{align*}
\text{tenv} & \vdash \text{body} : t_{\text{res}} \quad [\text{var} = t_{\text{var}}]\text{tenv} \vdash \text{body} : t_{\text{res}} \\
\text{tenv} & \vdash (\text{proc-exp } \text{var } t_{\text{var}} \text{ body}) : (t_{\text{var}} \rightarrow t_{\text{res}})
\end{align*}
\]

(proc-exp (var var-type body)
  (let ((result-type
        (type-of body
        (extend-tenv var var-type tenv))))
    (proc-type var var-type result-type)))
\[ tenv \vdash \text{rator} : (t_1 \rightarrow t_2) \quad tenv \vdash \text{rand} : t_1 \]
\[ tenv \vdash \text{(call-exp rator rand)} : t_2 \]

(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)))))))\]
\[
\begin{align*}
[p = (t \mapsto t_{res})]tenv &\vdash e_{proc-body} : t_{res} \\
[p = (t \mapsto t_{res})]tenv &\vdash e_{letrec-body} : t \\
tenv &\vdash (letrec-exp \ t_{res} \ p \ (var : t_{var}) = e_{proc-body} \ e_{letrec-body}) : t
\end{align*}
\]

(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
(proctype
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

```plaintext
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

```plaintext
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.

\[
\text{Optional-type ::= } \tau \\
\text{no-type ()}
\]

\[
\text{Optional-type ::= Type} \\
a-type (ty)
\]

\[
\text{Expression ::= proc (Identifier : Optional-type) Expression} \\
\text{proc-exp (var otype body)}
\]

\[
\text{Expression ::= letrec} \\
\text{Optional-type Identifier (Identifier : Optional-type) = Expression} \\
in Expression \\
\text{letrec-exp} \\
\text{(proc-result-otype proc-name} \\
\text{bound-var bound-var-otype proc-body} \\
\text{letrec-body)}
\]

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:

\[
\begin{align*}
\text{(diff-exp } e_1 e_2) & : t_{e_1} = \text{int} \\
& t_{e_2} = \text{int} \\
& t(\text{diff-exp } e_1 e_2) = \text{int} \\
\text{(zero?-exp } e_j) & : t_{e_j} = \text{int} \\
& t(\text{zero?-exp } e_j) = \text{bool} \\
\text{(if-exp } e_1 e_2 e_3) & : t_{e_1} = \text{bool} \\
& t_{e_2} = t(\text{if-exp } e_1 e_2 e_3) \\
& t_{e_3} = t(\text{if-exp } e_1 e_2 e_3) \\
\text{(proc-exp } var \text{ body}) & : t(\text{proc-exp } var \text{ body}) = (t_{\text{var}} \rightarrow t_{\text{body}}) \\
\text{(call-exp } rator \text{ rand}) & : t_{\text{erator}} = (t_{\text{rand}} \rightarrow t(\text{call-exp } rator \text{ rand}))
\end{align*}
\]

- The first rule says that the arguments and the result of a \text{diff-exp} must all be of type \text{int}.
- The second rule says that the argument of a \text{zero?-exp} must be an \text{int}, and its result is a \text{bool}.
- The third rule says that in an \text{if} expression, the test must be of type \text{bool}, and that the types of the two alternatives must be the same as the type of the entire \text{if} expression.
- The fourth rule says that the type of a \text{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

\[(\text{proc } x_1 \ldots x_n \ e) : t_{\text{proc } x_1 \ldots x_n} e = (t_{x_1} \cdots t_{x_n}) e \rightarrow te\]

\[(\text{call-exp } e_0 \ e_1 \ldots e_n) : t_{e_0} = (t_{e_1} \cdots t_{e_n}) \rightarrow t_{(\text{call-exp } e_0 \ e_1 \ldots e_n)}\]
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 \( \text{proc}(f)\text{proc}(x)-((f \ 3),(f \ x)) \). We begin by making a table of all the bound variables and applications in this expression, and assigning a type variable to each one.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f )</td>
<td>( t_f )</td>
</tr>
<tr>
<td>( x )</td>
<td>( t_x )</td>
</tr>
<tr>
<td>( \text{proc}(f)\text{proc}(x)-((f \ 3),(f \ x)) )</td>
<td>( t_0 )</td>
</tr>
<tr>
<td>( \text{proc}(x)-((f \ 3),(f \ x)) )</td>
<td>( t_1 )</td>
</tr>
<tr>
<td>( -((f \ 3),(f \ x)) )</td>
<td>( t_2 )</td>
</tr>
<tr>
<td>( (f \ 3) )</td>
<td>( t_3 )</td>
</tr>
<tr>
<td>( (f \ x) )</td>
<td>( t_4 )</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{proc}(f)\text{proc}(x)-((f \ 3),(f \ x)) )</td>
<td>1. ( t_0 = t_f \rightarrow t_1 )</td>
</tr>
<tr>
<td>( \text{proc}(x)-((f \ 3),(f \ x)) )</td>
<td>2. ( t_1 = t_x \rightarrow t_2 )</td>
</tr>
<tr>
<td>( -((f \ 3),(f \ x)) )</td>
<td>3. ( t_3 = \text{int} )</td>
</tr>
<tr>
<td></td>
<td>4. ( t_4 = \text{int} )</td>
</tr>
<tr>
<td></td>
<td>5. ( t_2 = \text{int} )</td>
</tr>
<tr>
<td>( (f \ 3) )</td>
<td>6. ( t_f = \text{int} \rightarrow t_3 )</td>
</tr>
<tr>
<td>( (f \ x) )</td>
<td>7. ( t_f = t_x \rightarrow t_4 )</td>
</tr>
</tbody>
</table>

- 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 \( \text{proc}(x)-((f \ 3),(f \ x)) \).
- Equation 2 says that \( \text{proc}(x)-((f \ 3),(f \ x)) \) 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

\[
\begin{align*}
t_0 &= t_f \rightarrow t_1 \\
t_1 &= t_x \rightarrow t_2 \\
t_3 &= \text{int} \\
t_4 &= \text{int} \\
t_2 &= \text{int} \\
t_f &= \text{int} \rightarrow t_3 \\
t_f &= t_x \rightarrow t_4
\end{align*}
\]

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.

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0 = tf \rightarrow t_1$</td>
<td>$t_0 = tf \rightarrow t_1$</td>
</tr>
<tr>
<td>$t_1 = tx \rightarrow t_2$</td>
<td>$t_0 = tf \rightarrow t_1$</td>
</tr>
<tr>
<td>$t_3 = \text{int}$</td>
<td>$t_3 = \text{int}$</td>
</tr>
<tr>
<td>$t_4 = \text{int}$</td>
<td>$t_4 = \text{int}$</td>
</tr>
<tr>
<td>$t_2 = \text{int}$</td>
<td>$t_2 = \text{int}$</td>
</tr>
<tr>
<td>$tf = \text{int} \rightarrow t_3$</td>
<td>$tf = \text{int} \rightarrow t_3$</td>
</tr>
<tr>
<td>$tf = tx \rightarrow t_4$</td>
<td>$tf = tx \rightarrow t_4$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 = tx \rightarrow t_2$</td>
<td>$t_0 = tf \rightarrow t_1$</td>
</tr>
<tr>
<td>$t_3 = \text{int}$</td>
<td>$t_3 = \text{int}$</td>
</tr>
<tr>
<td>$t_4 = \text{int}$</td>
<td>$t_4 = \text{int}$</td>
</tr>
<tr>
<td>$t_2 = \text{int}$</td>
<td>$t_2 = \text{int}$</td>
</tr>
<tr>
<td>$tf = \text{int} \rightarrow t_3$</td>
<td>$tf = \text{int} \rightarrow t_3$</td>
</tr>
<tr>
<td>$tf = tx \rightarrow t_4$</td>
<td>$tf = tx \rightarrow t_4$</td>
</tr>
</tbody>
</table>

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
\[ t_3 = \text{int} \]
\[ t_4 = \text{int} \]
\[ t_2 = \text{int} \]
\[ t_f = \text{int} \rightarrow t_3 \]
\[ t_f = t_x \rightarrow t_4 \]

Substitution
\[ t_0 = t_f \rightarrow (t_x \rightarrow t_2) \]
\[ t_1 = t_x \rightarrow t_2 \]

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
\[ t_4 = \text{int} \]
\[ t_2 = \text{int} \]
\[ t_f = \text{int} \rightarrow t_3 \]
\[ t_f = t_x \rightarrow t_4 \]

Substitution
\[ t_0 = t_f \rightarrow (t_x \rightarrow t_2) \]
\[ t_1 = t_x \rightarrow t_2 \]
\[ t_3 = \text{int} \]

Equations
\[ t_2 = \text{int} \]
\[ t_f = \text{int} \rightarrow t_3 \]
\[ t_f = t_x \rightarrow t_4 \]

Substitution
\[ t_0 = t_f \rightarrow (t_x \rightarrow t_2) \]
\[ t_1 = t_x \rightarrow t_2 \]
\[ t_3 = \text{int} \]
\[ t_4 = \text{int} \]

Equations
\[ t_f = \text{int} \rightarrow t_3 \]
\[ t_f = t_x \rightarrow t_4 \]

Substitution
\[ t_0 = t_f \rightarrow (t_x \rightarrow \text{int}) \]
\[ t_1 = t_x \rightarrow \text{int} \]
\[ t_3 = \text{int} \]
\[ t_4 = \text{int} \]
\[ t_2 = \text{int} \]
Now, the next equation to be considered contains $t_3$, which is already bound to \texttt{int} in the substitution. So we substitute \texttt{int} for $t_3$ in the equation. We would do the same thing for any other type variables in the equation. We call this \textit{applying} the substitution to the equation.

$$
\begin{align*}
\text{Equations} & \quad \text{Substitution} \\
\texttt{tf} &= \texttt{int} \rightarrow \texttt{int} & t_0 &= (\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{tx} \rightarrow \texttt{int}) \\
\texttt{tf} &= \texttt{tx} \rightarrow \texttt{t4} & t_1 &= \texttt{tx} \rightarrow \texttt{int} \\
\end{align*}
$$

We move the resulting equation into the substitution and update the substitution as necessary; this time no updating takes place since $\texttt{tf}$ does not occur in the substitution.

$$
\begin{align*}
\text{Equations} & \quad \text{Substitution} \\
\texttt{tf} &= \texttt{tx} \rightarrow \texttt{t4} & t_0 &= (\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{tx} \rightarrow \texttt{int}) \\
\end{align*}
$$

The next equation, $\texttt{tf} = \texttt{tx} \rightarrow \texttt{t4}$, contains $\texttt{tf}$ and $\texttt{t4}$, which are bound in the substitution, so we apply the substitution to this equation. This gets

$$
\begin{align*}
\text{Equations} & \quad \text{Substitution} \\
\texttt{tf} &= \texttt{tx} \rightarrow \texttt{t4} & t_0 &= (\texttt{int} \rightarrow \texttt{int}) \\
\end{align*}
$$
If neither side of the equation is a variable, we can simplify, yielding two new equations.

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.

The final equation, \( \text{int} = \text{int} \), is always true, so we can discard it.
We have no more equations, so we are done. We conclude from this calculation that our original expression \( \text{proc}(f) \text{proc}(x) - ((f \ 3), (f \ x)) \) should be assigned the type

\[
(\text{int} \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int})
\]

This is reasonable: The first argument \( f \) must take an \( \text{int} \) argument because it is given 3 as an argument. It must produce an \( \text{int} \), because its value is used as an argument to the subtraction operator. And \( x \) must be an \( \text{int} \), 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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>f</td>
<td>t_f</td>
</tr>
<tr>
<td>proc(f)(f 11)</td>
<td>t_0</td>
</tr>
<tr>
<td>(f 11)</td>
<td>t_1</td>
</tr>
</tbody>
</table>

Next we write down the equations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc(f)(f 11)</td>
<td>t_0 = t_f \rightarrow t_1</td>
</tr>
<tr>
<td>(f 11)</td>
<td>t_f = \text{int} \rightarrow t_1</td>
</tr>
</tbody>
</table>

And next we solve:

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_0 = t_f \rightarrow t_1</td>
<td></td>
</tr>
<tr>
<td>t_f = \text{int} \rightarrow t_1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_f = \text{int} \rightarrow t_1</td>
<td></td>
</tr>
<tr>
<td>t_0 = t_f \rightarrow t_1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_0 = (\text{int} \rightarrow t_1) \rightarrow t_1</td>
<td></td>
</tr>
<tr>
<td>t_f = \text{int} \rightarrow t_1</td>
<td></td>
</tr>
</tbody>
</table>

This means that we can assign proc(f)(f 11) the type (\text{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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>t_x</td>
</tr>
<tr>
<td>if x then -(x,1) else 0</td>
<td>t_0</td>
</tr>
<tr>
<td>-(x,1)</td>
<td>t_1</td>
</tr>
</tbody>
</table>

We then generate the equations

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>if x then -(x,1) else 0</td>
<td>t_x = bool</td>
</tr>
<tr>
<td></td>
<td>t_1 = t_0</td>
</tr>
<tr>
<td></td>
<td>int = t_0</td>
</tr>
<tr>
<td>-(x,1)</td>
<td>t_x = int</td>
</tr>
<tr>
<td></td>
<td>t_1 = int</td>
</tr>
</tbody>
</table>

Processing these equations as we did before, we get

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_x = bool</td>
<td>t_x = bool</td>
</tr>
<tr>
<td>t_1 = t_0</td>
<td></td>
</tr>
<tr>
<td>int = t_0</td>
<td></td>
</tr>
<tr>
<td>t_x = int</td>
<td></td>
</tr>
<tr>
<td>t_1 = int</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>t_1 = t_0</td>
<td></td>
</tr>
<tr>
<td>int = t_0</td>
<td></td>
</tr>
<tr>
<td>t_x = int</td>
<td></td>
</tr>
<tr>
<td>t_1 = int</td>
<td></td>
</tr>
</tbody>
</table>
Since \( t_x \) is already bound in the substitution, we apply the current substitution to the next equation, getting

\[
\begin{align*}
\text{Equations} & \quad \text{Substitution} \\
\text{int} &= t_0 & \quad t_x = \text{bool} \\
t_x &= \text{int} & \quad t_1 = t_0 \\
t_1 &= \text{int} & \\
\end{align*}
\]

Oops! We have inferred from these equations that \( \text{bool} = \text{int} \). So in any solution of these equations, \( \text{bool} = \text{int} \). But \( \text{bool} \) and \( \text{int} \) 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.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>proc (f) zero?((f f))</td>
<td>$t_0$</td>
</tr>
<tr>
<td>f</td>
<td>$t_f$</td>
</tr>
<tr>
<td>zero?((f f))</td>
<td>$t_1$</td>
</tr>
<tr>
<td>(f f)</td>
<td>$t_2$</td>
</tr>
</tbody>
</table>

And we solve as usual:

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_0 = t_f \rightarrow t_1$</td>
<td></td>
</tr>
<tr>
<td>$t_1 = \text{bool}$</td>
<td></td>
</tr>
<tr>
<td>$t_2 = \text{int}$</td>
<td></td>
</tr>
<tr>
<td>$t_f = t_f \rightarrow t_2$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 = \text{bool}$</td>
<td>$t_0 = t_f \rightarrow t_1$</td>
</tr>
<tr>
<td>$t_2 = \text{int}$</td>
<td></td>
</tr>
<tr>
<td>$t_f = t_f \rightarrow t_2$</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Equations</th>
<th>Substitution</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_2 = \text{int}$</td>
<td>$t_0 = t_f \rightarrow \text{bool}$</td>
</tr>
<tr>
<td>$t_f = t_f \rightarrow t_2$</td>
<td>$t_1 = \text{bool}$</td>
</tr>
</tbody>
</table>
Equations
\[ t_f = t_f \rightarrow t_2 \]

Substitution
\[ t_0 = t_f \rightarrow \text{bool} \\
    t_1 = \text{bool} \\
    t_2 = \text{int} \]

Equations
\[ t_f = t_f \rightarrow \text{int} \]

Substitution
\[ t_0 = t_f \rightarrow \text{bool} \\
    t_1 = \text{bool} \\
    t_2 = \text{int} \]

Now we have a problem. We’ve now inferred that \( t_f = t_f \rightarrow \text{Int} \). 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

\[
\text{proc}(f) \ \text{proc}(g) \ \text{proc}(x) \ \text{if} \ (g \ x) \ \text{then} \ (f \ g) \ \text{else} \ 27
\text{proc}(h) \ (h \ 11))
\]
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

\[
Type ::= \%tvar-type Number
\]

\[\text{tvar-type (serial-number)}\]

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

\textbf{apply-one-subst} : type * tvar * type -> type

\textbf{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 \(ty0[\text{tvar=ty1}]\)

\[
\text{(define apply-one-subst}
\]

\[
\text{(lambda (ty0 tvar ty1)
}\]

\[
\text{(cases type ty0
}\]

\[
\text{(proc-type (arg-type result-type)
}\]

\[
\text{(proc-type
}\]

\[
\text{(apply-one-subst arg-type tvar ty1)
}\]

\[
\text{(apply-one-subst result-type tvar ty1))}
\]

\[
\text{(tvar-type (sn)
}\]

\[
\text{(if (equal? ty0 tvar) ty1 ty0))
}\]

\[
\text{(else ty0)))}
\]

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 \textit{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 \texttt{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 auxiliary procedure \texttt{apply-subst-to-tvar-type} to look up the type variable in the substitution. This procedure uses Scheme procedure \texttt{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.

\textbf{apply-subst-to-tvar-type} : tvar-type * subst -> type

\begin{verbatim}
(define apply-subst-to-tvar-type
  (lambda (tvar-type subst)
    (let ((tmp (assoc tvar-type subst)))
      (if tmp
        (cdr tmp)
        tvar-type))))
\end{verbatim}

\textbf{apply-subst-to-type} : type * subst -> type

\begin{verbatim}
(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))))
\end{verbatim}
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 σ[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,

\[
\begin{pmatrix}
  tv_1 = t_1 \\
  \vdots \\
  tv_n = t_n
\end{pmatrix}
\]

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

This definition has the property that for any type t,

\[(tσ)[tv = t'] = t(σ[tv = t'])\]

This is a kind of associativity property.

The implementation of extend-subst follows the picture above. It substitutes t₀ for tv₀ in all of the existing bindings in σ₀, and then adds the binding for t₀.

\[
\begin{align*}
\text{(define empty-subst (lambda () '()))} \\
\text{extend-subst : subst * tvar * type -> subst} \\
\text{usage: tvar not already bound in subst.}
\end{align*}
\]

\[
\begin{align*}
\text{(define extend-subst} \\
\text{  (lambda (subst tvar ty))} \\
\text{  (cons} \\
\text{    (cons tvar ty)} \\
\text{    (map} \\
\text{      (lambda (p))} \\
\text{        (let ((tvar-type1 (car p))} \\
\text{          (type1 (cdr p))))} \\
\text{          (cons} \\
\text{            tvar-type1} \\
\text{            (apply-one-subst type1 tvar ty)))))} \\
\text{  subst)))))}
\end{align*}
\]
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 \texttt{unifier}. The unifier performs one step of the inference procedure outlined above: It takes two types, \( \text{ty1} \) and \( \text{ty2} \), a substitution \( \text{subst} \) that satisfies the no-occurrence invariant, and an expression \( \text{exp} \). It returns the substitution that results from adding \( \text{ty1} = \text{ty2} \) to \( \text{subst} \). This substitution will also satisfy the no-occurrence invariant. If adding \( \text{ty1} = \text{ty2} \) yields an inconsistency, then it reports an error, and blames the expression \( \text{exp} \). This is typically the expression that gave rise to the equation \( \text{ty1} = \text{ty2} \).

Think of the substitution as a \textit{store}, and an unknown type is a \textit{reference} into that store. \texttt{unifier} produces the new store that is obtained by adding \( \text{ty1} = \text{ty2} \) 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.

```scheme
(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 $ty_1$ and $ty_2$.

• If the resulting types are the same, we return immediately. This corresponds to the step of deleting a trivial equation above.

• If $ty_1$ 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 $ty_1 = ty_2$ to the substitution. But we need to perform the occurrence check, so that the no-occurrence invariant is preserved.

• If $ty_2$ is an unknown type, we do the same thing, reversing the roles of $ty_1$ and $ty_2$.

• If neither $ty_1$ nor $ty_2$ 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 $ty_1$ and $ty_2$ 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

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

\begin{verbatim}
(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))))
\end{verbatim}
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 \textit{store}, and a type variable as \textit{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 \texttt{unifier}.

```scheme
;; answer = type * subst

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

\textbf{type-of-program} : program \rightarrow 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

(definition type-of
  (lambda (exp tenv subst)
    (cases expression exp
      (const-exp (num) (an-answer (int-type) subst))
      (zero?-exp e1
        (let ((t_int (int-type subst1 exp))
          (t_bool (bool-type subst2)))
          (an-answer t_int))
      (diff-exp e1 e2
        (let ((t_int (int-type subst1))
          (t_int (int-type subst2))
          (t_int (int-type subst2))
          (an-answer t_int))))))))
(if-exp \( e_1 \ e_2 \ e_3 \))
\( t_{e_1} \) = bool
\( t_{e_2} \) = \( t_((\text{if-exp} \ e_1 \ e_2 \ e_3)) \)
\( t_{e_3} \) = \( t_((\text{if-exp} \ e_1 \ e_2 \ e_3)) \)

(if-exp \( \text{exp1} \ \text{exp2} \ \text{exp3} \))

(cases answer (type-of \text{exp1} tenv subst))

(an-answer \( \text{ty1} \ subst \))

(let ((subst

  (unifier \text{ty1} (bool-type) subst \text{exp1})))

  (cases answer (type-of \text{exp2} tenv subst))

  (an-answer \( \text{ty2} \ subst \))

(cases answer (type-of \text{exp3} tenv subst))

(an-answer \( \text{ty3} \ subst \))

(let ((subst

  (unifier \text{ty2} \text{ty3} subst \text{exp})))

  (an-answer \text{ty2} subst)))))

(var-exp \( \text{var} \))

(an-answer (apply-tenv tenv \text{var} subst))

(let-exp \( \text{var} \ \text{exp1} \ \text{body} \))

(cases answer (type-of \text{exp1} tenv subst))

(an-answer \( \text{exp1-type} \ subst \))

(type-of \text{body}

  (extend-tenv \text{var} \text{exp1-type} tenv)

  subst)))

45
(proc-exp var otype body) :  \( t_{\text{proc-exp var body}} = (t_{\text{var}} \rightarrow t_{\text{body}}) \)

(\begin{align*}
(\text{proc-exp (var otype body)}) \\
\quad \text{(let ((\text{var-type (otype->type otype)}))} \\
\quad \quad \text{(cases \text{answer} (type-of body}} \text{)} \\
\quad \quad \quad \text{(extend-tenv var var-type tenv subst)} \\
\quad \quad \text{(an-answer (result-type subst))} \\
\quad \text{(an-answer}} \\
\quad \quad \text{(proc-type var-type result-type subst))})
\end{align*})

(call-exp rator rand) :  \( t_{\text{erator}} = (t_{\text{rand}} \rightarrow t_{\text{(call-exp rator rand)}}) \)

(\begin{align*}
(\text{call-exp (erator rand)}) \\
\quad \text{(let ((\text{result-type (fresh-tvar-type)}))} \\
\quad \text{(cases \text{answer} (type-of rator tenv subst))} \\
\quad \quad \text{(an-answer (erator-type subst))} \\
\quad \text{(cases \text{answer} (type-of rand tenv subst))} \\
\quad \quad \text{(an-answer (rand-type subst))} \\
\quad \text{(let ((\text{subst}} \text{}}} \\
\quad \quad \text{(unifier \text{egrator-type}} \\
\quad \quad \text{(proc-type}} \\
\quad \quad \quad \text{(rand-type result-type}) \\
\quad \quad \quad \text{subst} \\
\quad \quad \quad \text{exp})) \\
\quad \quad \quad \text{(an-answer result-type subst))})
\end{align*})

46
\begin{align*}
\text{letrec-exp } & \quad t_{\text{proc-result}} \ p (x : t_{\text{var}}) = e_{\text{proc-body}} \ e_{\text{letrec-body}} \\
& \quad t_p = t_x \to t_{e_{\text{proc-body}}} \\
& \quad t_{e_{\text{letrec-body}}} = t_{\text{letrec-exp } t_{\text{proc-result}} \ p (x : t_{\text{var}}) = e_{\text{proc-body}} \ e_{\text{letrec-body}}}
\end{align*}

\begin{align*}
\text{letrec-exp } & \quad \text{(proc-result-otype \ proc-name} \\
& \quad \text{bound-var bound-var-otype} \\
& \quad \text{proc-body} \\
& \quad \text{letrec-body}) \\
& \quad \text{(let ((proc-result-type} \\
& \quad \text{(otype->type proc-result-otype))} \\
& \quad \text{(proc-var-type} \\
& \quad \text{(otype->type bound-var-otype)))} \\
& \quad \text{(let ((tenv-for-letrec-body} \\
& \quad \text{(extend-tenv} \\
& \quad \text{proc-name} \\
& \quad \text{(proc-type} \\
& \quad \text{(proc-var-type proc-result-type)} \\
& \quad \text{tenv)})}) \\
& \quad \text{(cases answer (type-of proc-body} \\
& \quad \text{(extend-tenv} \\
& \quad \text{bound-var proc-var-type} \\
& \quad \text{tenv-for-letrec-body} \\
& \quad \text{subst}) \\
& \quad \text{(an-answer (proc-body-type subst) } \\
& \quad \text{(let ((subst} \\
& \quad \text{(unifier} \\
& \quad \text{proc-body-type} \\
& \quad \text{proc-result-type} \\
& \quad \text{subst} \\
& \quad \text{proc-body)})) \\
& \quad \text{(type-of letrec-body} \\
& \quad \text{tenv-for-letrec-body} \\
& \quad \text{subst)))))})
\end{align*}
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 \texttt{proc (x) x}, it might generate any of the external forms \( (\text{ty}_1 \to \text{ty}_1) \) or \( (\text{ty}_2 \to \text{ty}_2) \) or \( (\text{ty}_3 \to \text{ty}_3) \), 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 \((\text{ty}_3 \to \text{ty}_4)\) or \((\text{int} \to \text{ty}_{17})\).

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 \texttt{ty}_1. We can then compare the renumbered types with \texttt{equal?}.

To systematically rename each unknown type, we construct a substitution with \texttt{canonical-subst}. This is a straightforward recursion, with \texttt{table} playing the role of an accumulator. The length of \texttt{table} tells us how many distinct unknown types we have found, so we can use its length to give the number of the “next” \texttt{ty} symbol. This is similar to the way we used \texttt{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:

```plaintext
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 $(\forall t)(t \to list t \to list t)$, and `map` is of type $(\forall s,t)((s \to t) \to list s \to list t)$.

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 $(\forall t)(bool \to t \to t \to t)$.

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.

```plaintext
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

\[
(fst (pair x y)) = x \\
(snd (pair x y)) = y
\]

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

```plaintext
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

```plaintext
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,

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

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
\(\ty \to ((\tx \times\ \ty \to\ u) \to\ u)\)

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:

\begin{itemize}
  \item \texttt{add1((fst (pairx true)))}: from this we can deduce that \tx must be \texttt{int}.
  \item \texttt{if (fst (pairx true)) then 11 else 22}: from this we can deduce that \tx must be \texttt{bool}.
\end{itemize}

If we allowed \tx to vary, we could write
\begin{verbatim}
if (fst (pairx true)) then 11 else add1((fst (pairx true)))
\end{verbatim}

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
\(\forall\ \ty,\ u (\ty \to ((\tx \times\ \ty \to\ u) \to\ u))\)

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.

\[
\begin{align*}
tenv(x) &= (\text{all } x_1, \ldots x_n)t \\
tv_1, \ldots, tv_n \text{ fresh} \\
\text{(type-of-expression} \langle\langle x\rangle\rangle \text{tenv}) &= t[tv_1/x_1, \ldots ,tv_n/x_n] \\
\end{align*}
\]

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