Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

The MIT Press. (c) 2008. Copying Prohibited.

Reprinted for Ryan Hoffman, Library

Ryan.Hoffman@Colorado.EDU

Reprinted with permission as a subscription benefit of **Skillport**,

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 7: Types

Overview

We've seen how we can use interpreters to model the run-time behavior of programs. Now we'd like to use the same technology to *analyze* or *predict* the behavior of programs without running them.

We've already seen some of this: our lexical-address translator predicts at analysis time where in the environment each variable will be found at run time. Further, the actual translator looked like an interpreter, except that instead of passing around an environment, we passed around a static environment.

Our goal is to analyze a program to predict whether evaluation of a program is *safe*, that is, whether the evaluation will proceed without certain kinds of errors. Exactly what is meant by safety, however, may vary from language to language. If we can guarantee that evaluation is safe, we will be sure that the program satisfies its contract.

In this chapter, we will consider languages that are similar to LETREC in chapter 3. For these languages we say that an evaluation is *safe* if and only if:

1. For every evaluation of a variable *var*, the variable is bound.
2. For every evaluation of a difference expression (diff-exp exp_1 exp_2), the values of exp_1 and exp_2 are both num-vals.
3. For every evaluation of an expression of the form (zero?-exp exp_1), the value of exp_1 is a num-val.
4. For every evaluation of a conditional expression (if-exp exp_1 exp_2 exp_3), the value of exp_1 is a bool-val.
5. For every evaluation of a procedure call (call-exp *rator rand*), the value of *rator* is a proc-val.

These conditions assert that each operator is performed only on operands of the correct type. We therefore call violations of these conditions *type errors*.

A safe evaluation may still fail for other reasons: division by zero, taking the car of an empty list, etc. We do not include these as part of our definition of safety because predicting safety for these conditions is much harder than guaranteeing the conditions listed above. Similarly, a safe evaluation may run infinitely. We do not include nontermination as part of safety because checking for termination is also very difficult (indeed, it is undecidable in general). Some languages have type systems that guarantee conditions stronger than the ones above, but those are more complex than the ones we consider here.

Our goal is to write a procedure that looks at the program text and either accepts or rejects it. Furthermore, we would like our analysis procedure to be conservative: if the analysis accepts the program, then we can be sure evaluation of the program will be 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*.

An analysis that rejected every program would still be sound, so we also want our analysis to accept a large set of programs. The analyses in this chapter will accept enough programs to be useful.

Here are some examples of programs that should be rejected or accepted by our analysis:

if 3 then 88 else 99	reject: non-boolean test
proc (x) (x 3)	reject: non-proc-val rator
proc (x) (3 x)	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 = zero?(0) in -(3, x)	reject: non-integer argument to a diff-exp
`(proc (x) -(3,x) zero?(0))`	reject: same as preceding example

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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 nonterminating but safe

Although the evaluation of the last example does not terminate, the evaluation is safe by the definition given above, so our analysis is permitted to accept it. As it turns out, our analysis will accept it, because the analysis is not fine enough to determine that this program does not halt.

7.1 Values and Their Types

Since the safety conditions talk only 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 can not 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 one-argument procedures, but dealing with multiargument procedures, as in exercise 3.33, is straightforward: it requires some additional work but does not require any new ideas.

Grammar for Types

$$\mathit{Type} ::= \texttt{int}$$
$$\boxed{\texttt{int-type ()}}$$

$$\mathit{Type} ::= \texttt{bool}$$
$$\boxed{\texttt{bool-type ()}}$$

$$\mathit{Type} ::= (\mathit{Type}\ \texttt{->}\ \mathit{Type})$$
$$\boxed{\texttt{proc-type (arg-type result-type)}}$$

To see how this type system works, let's look at some examples.

Examples of Values and Their Types

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) since, when given an integer, it returns an integer.

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

The value of proc (x) if x then 11 else 22 has type (bool -> int), since 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, since 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)), since when given a boolean, it returns a

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

procedure from booleans to integers.

The value of proc (f) if (f 3) then 11 else 22 has type ((int -> bool) -> int), since 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*, since 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*, since when given a procedure of type (*t* -> *t*), it returns another procedure that, when given an argument of type *t*, returns a value of type *t*.

We can explain these examples by the following definition.

Definition 7.1.1 *The property of an expressed value v being of type t is defined by induction on t:*

- *An expressed value is of type* int *if and only if it is a* num-val*.*
- *It is of type* bool *if and only if it is a* bool-val*.*
- *It is of type* ($t_1 \to t_2$) *if and only if 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 happens:*
  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*."

This is a definition by induction on *t*. It depends, however, on the set of type errors being defined independently, as we did above.

In this system, a value *v* can be of more than one type. For example, the value of proc (x) x is of type ($t \to t$) for any type *t*. Some values may have no type, like the value of proc (x) if x then 11 else zero?(11).

Exercise 7.1 [*] Below is a list of closed expressions. Consider the value of each expression. For each value, what type or types does it have? Some of the values may have no type that is describable in our type language.

```
 1. proc (x) -(x,3)
 2. proc (f) proc (x) -((f x), 1)
 3. proc (x) x
 4. proc (x) proc (y) (x y).
 5. proc (x) (x 3)
 6. proc (x) (x x)
 7. proc (x) if x then 88 else 99
 8. proc (x) proc (y) if x then y else 99
 9. (proc (p) if p then 88 else 99
     33)
10. (proc (p) if p then 88 else 99
     proc (z) z)
11. proc (f)
     proc (g)
      proc (p)
       proc (x) if (p (f x)) then (g 1) else -((f x),1)
12. proc (x)
     proc(p)
      proc (f)
       if (p x) then -(x,1) else (f p)
13. proc (f)
     let d = proc (x)
              proc (z) ((f (x x)) z)
     in proc (n) ((f (d d)) n)
```

Exercise 7.2 [**] Are there any expressed values that have exactly two types according to definition 7.1.1?

Exercise 7.3 [**] For the language LETREC, is it decidable whether an expressed value *val* is of type *t*?

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

7.2 Assigning a Type to an Expression

So far, we've dealt only with the types of expressed values. In order to analyze programs, we need to write a procedure that takes an expression and predicts the type of its value.

More precisely, our goal is to 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.

If we can assign an expression to a type, we say that the expression is *well-typed*; otherwise we say it is *ill-typed* or *has no type*.

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 down a set of rules that type-of should follow. Assume that *tenv* is a *type environment* mapping each variable to its type. Then we should have:

Simple Typing Rules

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$$(\texttt{type-of}\ (\texttt{const-exp}\ num)\ tenv) = \texttt{int}$$

$$(\texttt{type-of}\ (\texttt{var-exp}\ var)\ tenv) = tenv(var)$$

$$\frac{(\texttt{type-of}\ exp_1\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{zero?-exp}\ exp_1)\ tenv) = \texttt{bool}}$$

$$\frac{(\texttt{type-of}\ exp_1\ tenv) = \texttt{int} \quad (\texttt{type-of}\ exp_2\ tenv) = \texttt{int}}{(\texttt{type-of}\ (\texttt{diff-exp}\ exp_1\ exp_2)\ tenv) = \texttt{int}}$$

$$\frac{(\texttt{type-of}\ exp_1\ tenv) = t_1 \quad (\texttt{type-of}\ body\ [var=t_1]tenv) = t_2}{(\texttt{type-of}\ (\texttt{let-exp}\ var\ exp_1\ body)\ tenv) = t_2}$$

$$\frac{\begin{array}{l}(\texttt{type-of}\ exp_1\ tenv) = \texttt{bool}\\ (\texttt{type-of}\ exp_2\ tenv) = t\\ (\texttt{type-of}\ exp_3\ tenv) = t\end{array}}{(\texttt{type-of}\ (\texttt{if-exp}\ exp_1\ exp_2\ exp_3)\ tenv) = t}$$

$$\frac{(\texttt{type-of}\ rator\ tenv) = (t_1 \to t_2) \quad (\texttt{type-of}\ rand\ tenv) = t_1}{(\texttt{type-of}\ (\texttt{call-exp}\ rator\ rand)\ tenv) = t_2}$$

If we evaluate an expression *exp* of type *t* in a suitable environment, we know not only that its value is of type *t*, but we also know something about the history of that value. Because the evaluation of *exp* is guaranteed to be safe, we know that the value of *exp* was constructed only by operators that are legal for type *t*. This point of view will be helpful when we consider data abstraction in more detail in chapter 8.

What about procedures? If proc(*var*)*body* has type ($t_1 \to t_2$), then it is intended to be called on an argument of type t_1. When *body* is evaluated, the variable *var* will be bound to a value of type t_1.

This suggests the following rule:

$$\frac{(\texttt{type-of}\ body\ [var=t_1]tenv) = t_2}{(\texttt{type-of}\ (\texttt{proc-exp}\ var\ body)\ tenv) = (t_1 \to t_2)}$$

This rule is sound: if type-of makes correct predictions about *body*, then it makes correct predictions about (proc-exp *var body*).

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.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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 of the bound variables based on how the variables are used in the program. If the language is carefully designed, the type-checker can infer most or all of these types.

We will study each of these in turn.

Exercise 7.4 [*] Using the rules of this section, write derivations, like the one on page 5, that assign types for proc (x) x and proc (x) proc (y) (x y). Use the rules to assign at least two types for each of these terms. Do the values of these expressions have the same types?

7.3 CHECKED: A Type-Checked Language

Our new language will be the same as LETREC, except that we require the programmer to include the types of all bound variables. For letrec-bound variables, we also require the programmer to specify the result type of the procedure as well.

Here are some example programs in CHECKED.

```
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), since it is a procedure that takes an integer and returns an integer.

To define the syntax of this language, we change the productions for proc and letrec expressions.

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
  (p-result-type p-name b-var b-var-type
   p-body
   letrec-body)
```

For a proc expression with the type of its bound variable specified, the rule becomes

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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

What about letrec? A typical letrec looks like

```
letrec
   t_res p (var : t_var) = e_proc-body
in e_letrec-body
```

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

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

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

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

$$tenv_{proc\text{-}body} = [var = t_{var}]\,tenv_{letrec\text{-}body}$$

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

Writing this down as a rule, we get:

$$\frac{(\texttt{type-of}\ e_{proc\text{-}body}\ [var{=}t_{var}]\,[p = (t_{var} \rightarrow t_{res})]\,tenv) = t_{res} \qquad (\texttt{type-of}\ e_{letrec\text{-}body}\ [p = (t_{var} \rightarrow t_{res})]\,tenv) = t}{(\texttt{type-of}\ \texttt{letrec}\ t_{res}\ p\ (var : t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}\ tenv) = t}$$

Now we have written down all the rules, so we are ready to implement a type checker for this language.

7.3.1 The Checker

We will need to compare types for equality. We do this with the procedure check-equal-type!, which compares two types and reports an error unless they are equal. check-equal-type! takes a third argument, which is the expression that we will blame if the types are unequal.

```
check-equal-type! : Type × Type × Exp → Unspecified
(define check-equal-type!
  (lambda (ty1 ty2 exp)
    (if (not (equal? ty1 ty2))
      (report-unequal-types ty1 ty2 exp))))

report-unequal-types : Type × Type × Exp → Unspecified
(define report-unequal-types
  (lambda (ty1 ty2 exp)
    (eopl:error 'check-equal-type!
      "Types didn't match: ~s != ~a in~%~a"
      (type-to-external-form ty1)
      (type-to-external-form ty2)
      exp)))
```

We never use the value of a call to check-equal-type!; thus a call to check-equal-type! is executed for effect only, like the setref expressions in section 4.2.2.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

The procedure report-unequal-types uses type-to-external-form, which converts a type back into a list that is easy to read.

```
type-to-external-form : Type → List
(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 did for interpreters in chapter 3. The result is shown in figures 7.1–7.3.

```
Tenv = Var → Type

type-of-program : Program → Type
(define type-of-program
  (lambda (pgm)
    (cases program pgm
      (a-program (exp1) (type-of exp1 (init-tenv))))))
type-of : Exp × Tenv → Type
(define type-of
  (lambda (exp tenv)
    (cases expression exp
```

(type-of *num tenv*) = int

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

(type-of *var tenv*) = *tenv*(*var*)

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

$$\frac{\text{(type-of } e_1 \text{ } tenv\text{) = int} \quad \text{(type-of } e_2 \text{ } tenv\text{) = int}}{\text{(type-of (diff-exp } e_1 \text{ } e_2\text{) } tenv\text{) = int}}$$

```
      (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)))
```

$$\frac{\text{(type-of } e_1 \text{ } tenv\text{) = int}}{\text{(type-of (zero?-exp } e_1\text{) } tenv\text{) = bool}}$$

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

$$\frac{\begin{array}{l}\text{(type-of } e_1 \text{ } tenv\text{) = bool} \\ \text{(type-of } e_2 \text{ } tenv\text{) = } t \\ \text{(type-of } e_3 \text{ } tenv\text{) = } t\end{array}}{\text{(type-of (if-exp } e_1 \text{ } e_2 \text{ } e_3\text{) } tenv\text{) = } t}$$

```
      (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))
```

Figure 7.1: `type-of` for CHECKED

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$$\frac{\texttt{(type-of } e_1 \ tenv\texttt{)} = t_1 \qquad \texttt{(type-of } body \ [var{=}t_1]\,tenv\texttt{)} = t_2}{\texttt{(type-of (let-exp } var \ e_1 \ body\texttt{)} \ tenv\texttt{)} = t_2}$$

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

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

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

$$\frac{\texttt{(type-of } rator \ tenv\texttt{)} = (t_1 \rightarrow t_2) \qquad \texttt{(type-of } rand \ tenv\texttt{)} = t_1}{\texttt{(type-of (call-exp } rator \ rand\texttt{)} \ tenv\texttt{)} = t_2}$$

```
(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
        (report-rator-not-a-proc-type
          rator-type rator)))))
```

Figure 7.2: `type-of` for CHECKED

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$$\frac{\begin{array}{c}(\texttt{type-of}\ e_{proc\text{-}body}\ [var{=}t_{var}]\ [p = (t_{var} \rightarrow t_{res})]\ tenv) = t_{res} \\ (\texttt{type-of}\ e_{letrec\text{-}body}\ [p = (t_{var} \rightarrow t_{res})]\ tenv) = t\end{array}}{(\texttt{type-of letrec}\ t_{res}\ p\ (var : t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}\ tenv) = t}$$

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

Figure 7.3: `type-of` for CHECKED

Exercise 7.5 [**] Extend the checker to handle multiple let declarations, multiargument procedures, and multiple letrec declarations. You will need to add types of the form (t_1 * t_2 * ... * t_n -> t) to handle multiargument procedures.

Exercise 7.6 [*] Extend the checker to handle assignments (section 4.3).

Exercise 7.7 [*] Change the code for checking an if-exp so that if the test expression is not a boolean, the other expressions are not checked. Give an expression for which the new version of the checker behaves differently from the old version.

Exercise 7.8 [**] Add pairof types to the language. Say that a value is of type pairof t_1 * t_2 if and only if it is a pair consisting of a value of type t_1 and a value of type t_2. Add to the language the following productions:

Type ::= `pairof` *Type* `*` *Type*

```
pair-type (ty1 ty2)
```

Expression ::= `newpair` `(` *Expression* `,` *Expression* `)`

```
pair-exp (exp1 exp2)
```

Expression ::= `unpair` *Identifier* *Identifier* `=` *Expression*
`in` *Expression*

```
unpair-exp (var1 var2 exp body)
```

A pair expression creates a pair; an unpair expression (like exercise 3.18) binds its two variables to the two parts of the

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

expression; the scope of these variables is body. The typing rules for pair and unpair are:

$$\frac{\begin{array}{c}\texttt{(type-of } e_1 \ tenv\texttt{)} = t_1 \\ \texttt{(type-of } e_2 \ tenv\texttt{)} = t_2\end{array}}{\texttt{(type-of (pair-exp } e_1 \ e_2\texttt{) } tenv\texttt{)} = \texttt{pairof } t_1 \texttt{ * } t_2}$$

$$\frac{\begin{array}{c}\texttt{(type-of } e_{pair} \ tenv\texttt{)} = \texttt{(pairof } t_1 \ t_2\texttt{)} \\ \texttt{(type-of } e_{body} \ [var_1{=}t_1]\,[var_2{=}t_2]\,tenv\texttt{)} = t_{body}\end{array}}{\texttt{(type-of (unpair-exp } var_1 \ var_2 \ e_1 \ e_{body}\texttt{) } tenv\texttt{)} = t_{body}}$$

Extend CHECKED to implement these rules. In type-to-external-form, produce the list (pairof t_1 t_2) for a pair type.

Exercise 7.9 [**] Add listof types to the language, with operations similar to those of exercise 3.9. A value is of type listof *t* if and only if it is a list and all of its elements are of type *t*. Extend the language with the productions

Type ::= `listof` *Type*

```
list-type (ty)
```

Expression ::= `list (`*Expression* { `,` *Expression* }* `)`

```
list-exp (exp1 exps)
```

Expression ::= `cons (`*Expression* `,` *Expression* `)`

```
cons-exp (exp1 exp2)
```

Expression ::= `null? (`*Expression* `)`

```
null-exp (exp1)
```

Expression ::= `emptylist_`*Type*

```
emptylist-exp (ty)
```

with types given by the following four rules:

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$$\frac{\begin{array}{c}(\texttt{type-of}\ e_1\ tenv) = t \\ (\texttt{type-of}\ e_2\ tenv) = t \\ \vdots \\ (\texttt{type-of}\ e_n\ tenv) = t\end{array}}{(\texttt{type-of}\ (\texttt{list-exp}\ e_1\ (e_2\ \ldots\ e_n))\ tenv) = \texttt{listof}\ t}$$

$$\frac{\begin{array}{c}(\texttt{type-of}\ e_1\ tenv) = t \\ (\texttt{type-of}\ e_2\ tenv) = \texttt{listof}\ t\end{array}}{(\texttt{type-of}\ \texttt{cons}(e_1, e_2)\ tenv) = \texttt{listof}\ t}$$

$$\frac{(\texttt{type-of}\ e_1\ tenv) = \texttt{listof}\ t}{(\texttt{type-of}\ \texttt{null?}(e_1)\ tenv) = \texttt{bool}}$$

$$(\texttt{type-of}\ \texttt{emptylist}[t]\ tenv) = \texttt{listof}\ t$$

Although cons is similar to pair, it has a very different typing rule.

Write similar rules for car and cdr, and extend the checker to handle these as well as the other expressions. Use a trick similar to the one in exercise 7.8 to avoid conflict with proc-type-exp. These rules should guarantee that car and cdr are applied to lists, but they should not guarantee that the lists be non-empty. Why would it be unreasonable for the rules to guarantee that the lists be non-empty? Why is the type parameter in emptylist necessary?

Exercise 7.10 [**] Extend the checker to handle EXPLICIT-REFS. You will need to do the following:

- Add to the type system the types refto *t*, where *t* is any type. This is the type of references to locations containing a value of type *t*. Thus, if *e* is of type *t*, (newref *e*) is of type refto *t*.
- Add to the type system the type void. This is the type of the value returned by setref. You can't apply any operation to a value of type void, so it doesn't matter what value setref returns. This is an example of types serving as an information-hiding mechanism.
- Write down typing rules for newref, deref, and setref.
- Implement these rules in the checker.

Exercise 7.11 [**] Extend the checker to handle MUTABLE-PAIRS.

7.4 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.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

For our case study in type inference, we start with the language of CHECKED. We then change the language so that all the type expressions 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 at the end of zero?) indicates a place where a type expression must be inferred.

Since the type expressions are optional, we may 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 nonterminal, *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
  (p-result-otype p-name
   b-var b-var-otype p-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 *var*, let t_e or t_{var} 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 would be:

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

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

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

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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

(proc-exp *var body*) : $t_{\texttt{(proc-exp var body)}} = (t_{var} \to t_{body})$
(call-exp *rator rand*) : $t_{rator} = (t_{rand} \to t_{\texttt{(call-exp rator rand)}})$

- 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.

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, proc expressions, if expressions, and procedure calls 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.	$t_0 = t_f \to t_1$
proc(x)-((f 3),(f x))	2.	$t_1 = t_x \to t_2$
-((f 3),(f x))	3.	$t_3 = \text{int}$
	4.	$t_4 = \text{int}$
	5.	$t_2 = \text{int}$
(f 3)	6.	$t_f = \text{int} \to t_3$
(f x)	7.	$t_f = t_x \to t_4$

- 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).

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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

$t_0 = t_f \to t_1$
$t_1 = t_x \to t_2$
$t_3 = \text{int}$
$t_4 = \text{int}$
$t_2 = \text{int}$
$t_f = \text{int} \to t_3$
$t_f = t_x \to t_4$

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
$t_0 = t_f \to t_1$	
$t_1 = t_x \to t_2$	
$t_3 = \text{int}$	
$t_4 = \text{int}$	
$t_2 = \text{int}$	
$t_f = \text{int} \to t_3$	
$t_f = t_x \to t_4$	

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

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

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
$t_3 = \text{int}$	$t_0 = t_f \to (t_x \to t_2)$
$t_4 = \text{int}$	$t_1 = t_x \to t_2$
$t_2 = \text{int}$	
$t_f = \text{int} \to t_3$	
$t_f = t_x \to t_4$	

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
$t_4 = \text{int}$	$t_0 = t_f \to (t_x \to t_2)$

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Equations	Substitution
$t_2 = \text{int}$	$t_1 = t_x \to t_2$
$t_f = \text{int} \to t_3$	$t_3 = \text{int}$
$t_f = t_x \to t_4$	

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

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

Now, the next equation to be considered contains t_3, which is already bound to 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
$t_f = \text{int} \to \text{int}$	$t_0 = t_f \to (t_x \to \text{int})$
$t_f = t_x \to t_4$	$t_1 = t_x \to \text{int}$
	$t_3 = \text{int}$
	$t_4 = \text{int}$
	$t_2 = \text{int}$

We move the resulting equation into the substitution and update the substitution as necessary.

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

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

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

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

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Equations	Substitution
int = t_x	t_0 = (int → int) → (t_x → int)
int = int	t_1 = t_x → int
	t_3 = int
	t_4 = int
	t_2 = int
	t_f = int → 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
int = int	t_0 = (int → int) → (int → int)
	t_1 = int → int
	t_3 = int
	t_4 = int
	t_2 = int
	t_f = int → int
	t_x = int

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

Equations	Substitution
	t_0 = (int → int) → (int → int)
	t_1 = int → int
	t_3 = int
	t_4 = int
	t_2 = int
	t_f = int → int
	t_x = 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

```
((int → int) → (int → 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 → t_1
(f 11)	t_f = int → t_1

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

And next we solve:

Equations	Substitution
$t_0 = t_f \to t_1$	
$t_f = \text{int} \to t_1$	

Equations	Substitution
$t_f = \text{int} \to t_1$	$t_0 = t_f \to t_1$

Equations	Substitution
	$t_0 = (\text{int} \to t_1) \to t_1$
	$t_f = \text{int} \to t_1$

This means that we can assign proc (f) (f 11) the type $(\text{int} \to t_1) \to t_1$, for any choice of t_1. Again, this is reasonable: we can infer that f must be able to take 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	$t_x = \text{bool}$
	$t_1 = t_0$
	$\text{int} = t_0$
-(x,1)	$t_x = \text{int}$
	$t_1 = \text{int}$

Processing these equations as we did before, we get

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

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

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

Equations	Substitution

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Equations	Substitution
t_0 = int	t_x = bool
t_x = int	t_1 = t_0
t_1 = int	

Equations	Substitution
t_x = int	t_x = bool
t_1 = int	t_1 = int
	t_0 = int

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

Equations	Substitution
bool = int	t_x = bool
t_1 = int	t_1 = int
	t_0 = int

What has happened here? 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))	$t_0 = t_f \to t_1$
zero?((f f))	t_1 = bool
	t_2 = int
(f f)	$t_f = t_f \to t_2$

And we solve as usual:

Equations	Substitution
$t_0 = t_f \to t_1$	
t_1 = bool	
t_2 = int	
$t_f = t_f \to t_2$	

Equations	Substitution
t_1 = bool	$t_0 = t_f \to t_1$
t_2 = int	
$t_f = t_f \to t_2$	

Equations	Substitution
t_2 = int	$t_0 = t_f \to$ bool
$t_f = t_f \to t_2$	t_1 = bool

Equations	Substitution

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Equations	Substitution
$t_f = t_f \to t_2$	$t_0 = t_f \to \text{bool}$
	$t_1 = \text{bool}$
	$t_2 = \text{int}$

Equations	Substitution
$t_f = t_f \to \text{int}$	$t_0 = t_f \to \text{bool}$
	$t_1 = \text{bool}$
	$t_2 = \text{int}$

Now we have a problem. We've now inferred that $t_f = t_f \to \mathit{Int}$. 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.

Exercise 7.12 [*] Using the methods in this section, derive types for each of the expressions in exercise 7.1, or determine that no such type exists. As in the other examples of this section, assume there is a ? attached to each bound variable.

Exercise 7.13 [*] Write down a rule for doing type inference for let expressions. Using your rule, derive types for each of the following expressions, or determine that no such type exists.

1. let x = 4 in (x3)
2. let f = proc (z) z in proc (x) -((f x), 1)
3. let p = zero?(1) in if p then 88 else 99
4. let p = proc (z) z in if p then 88 else 99

Exercise 7.14 [*] What is wrong with this expression?

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

Exercise 7.15 [**] Write down a rule for doing type inference for a letrec expression. Your rule should handle multiple declarations in a letrec. Using your rule, derive types for each of the following expressions, or determine that no such type exists:

```
1. letrec ? f (x : ?)
           = if zero?(x) then 0 else -((f -(x,1)), -2)
   in f
2. letrec ? even (x : ?)
              = if zero?(x) then 1 else (odd -(x,1))
           ? odd (x : ?)
              = if zero?(x) then 0 else (even -(x,1))
   in (odd 13)
3. letrec ? even (odd : ?)
              = proc (x) if zero?(x)
```


Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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

Exercise 7.16 [***] Modify the grammar of INFERRED so that missing types are simply omitted, rather than marked with ?.

7.4.1 Substitutions

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

We represent type variables as an additional variant of the type data type. We do this using the same technique that we used for lexical addresses in section 3.7. We add to the grammar the production

$$\textit{Type} ::= \texttt{\%tvar-type}\ \textit{Number}$$

```
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. (apply-one-subst t_0 *tv* t_1) returns the type obtained by substituting t_1 for every occurrence of *tv* in t_0. This is some times written $t_0[tv = t_1]$.

```
apply-one-subst : Type × Tvar × Type → Type
(define apply-one-subst
  (lambda (ty0 tvar ty1)
    (cases type ty0
      (int-type () (int-type))
      (bool-type () (bool-type))
      (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)))))
```

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 pairs (type variable . type). The basic observer for substitutions is apply-subst-to-type. This walks through the type *t*, replacing each type variable by its binding in the substitution σ. If a variable is not bound in the substitution, then it is left unchanged. We write $t\sigma$ for the resulting type.

The implementation uses the Scheme procedure assoc to look up the type variable in the substitution. assoc 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. We write

```
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)
        (let ((tmp (assoc ty subst)))
          (if tmp
            (cdr tmp)
            ty))))))
```


Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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*,

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

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.

```
empty-subst : () → Subst
(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 ((oldlhs (car p))
                (oldrhs (cdr p)))
            (cons
              oldlhs
              (apply-one-subst oldrhs tvar ty))))
        subst))))
```

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

Exercise 7.17 [**] In our representation, extend-subst may do a lot of work if σ is large. Implement an alternate representation in which extend-subst is implemented as

```
(define extend-subst
  (lambda (subst tvar ty)
    (cons (cons tvar ty) subst)))
```

and the extra work is shifted to apply-subst-to-type, so that the property $t(\sigma[tv = t']) = (t\sigma)[tv = t']$ is still satisfied. For this definition of extend-subst, is the no-occurrence invariant needed?

Exercise 7.18 [**] Modify the implementation in the preceding exercise so that apply-subst-to-type computes the substitution for any type variable at most once.

7.4.2 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, t_1 and t_2, a substitution σ that satisfies the no-occurrence invariant, and an expression *exp*. It returns the substitution that results from adding $t_1 = t_2$ to σ. This will be the smallest extension of σ that unifies $t_1\sigma$ and $t_2\sigma$. This substitution will still satisfy the no-occurrence invariant. If adding $t_1 = t_2$ yields an inconsistency or violates the no-occurrence invariant, then the unifier reports an error, and blames the expression exp. This is typically the expression that gave rise to the equation $t_1 = t_2$.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

This is an algorithm for which cases gives awkward code, so we use simple predicates and extractors on types instead. The algorithm is shown in figure 7.4, and it works as follows:

- First, as we did above, we apply the substitution to each of the types t_1 and t_2.
- 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 $t_1 = t_2$ to the substitution. But we need to perform the occurrence check, so that the no-occurrence invariant is preserved. The call (no-occurrence? *tv t*) returns #t if and only if there is no occurrence of the type variable *tv* in *t* (figure 7.5).

```
no-occurrence? : Tvar × Type → Bool
(define no-occurrence?
  (lambda (tvar ty)
    (cases type ty
      (int-type () #t)
      (bool-type () #t)
      (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))))))
```

Figure 7.5: The occurrence check

- If t_2 is an unknown type, we do the same thing, reversing the roles of t_1 and t_2.
- If neither t_1 nor t_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 t_1 and t_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.

```
unifier : Type × Type × Subst × Exp → Subst
(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)
           (report-no-occurrence-violation ty1 ty2 exp)))
        ((tvar-type? ty2)
         (if (no-occurrence? ty2 ty1)
           (extend-subst subst ty2 ty1)
           (report-no-occurrence-violation 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 (report-unification-failure ty1 ty2 exp))))))
```

Figure 7.4: The unifier

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Here is another way of thinking about all this that is sometimes useful. The substitution is 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.

Last, we must implement the occurrence check. This is a straightforward recursion on the type, and is shown in figure 7.5.

Exercise 7.19 [*] We wrote: “If ty1 is an unknown type, then the no-occurrence invariant tells us that it is not bound in the substitution.” Explain in detail why this is so.

Exercise 7.20 [**] Modify the unifier so that it calls apply-subst-to-type only on type variables, rather than on its arguments.

Exercise 7.21 [**] We said the substitution is like a store. Implement the unifier, using the representation of substitutions from exercise 7.17, and keeping the substitution in a global Scheme variable, as we did in figures 4.1 and 4.2.

Exercise 7.22 [**] Refine the implementation of the preceding exercise so that the binding of each type variable can be obtained in constant time.

7.4.3 Finding the Type of an Expression

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

```
optype->type : OptionalType → Type
(define otype->type
  (lambda (otype)
    (cases optional-type otype
      (no-type () (fresh-tvar-type))
      (a-type (ty) ty))))

fresh-tvar-type : () → Type
(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.

```
type-to-external-form : Type → List
(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. We use the metaphor of a 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 as we did in exercise 4.12, by defining a new data type that contains the two values, and using that as the return value.

The definition of type-of is shown in figures 7.6–7.8. 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.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$Answer = Type \times Subst$

```
(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)))))
```

Figure 7.6: `type-of` for INFERRED, part 1

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

(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))))
```

Figure 7.7: `type-of` for INFERRED, part 2

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

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

```
(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 (body-type subst)
        (an-answer
          (proc-type var-type body-type)
          subst)))))
```

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

```
(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)))))))))
```

Figure 7.8: `type-of` for INFERRED, part 3

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 (tvar1 -> tvar1) or (tvar2 -> tvar2) or (tvar3 -> tvar3), 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 (tvar3 -> tvar4) or (int -> tvar17).

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? (figures 7.10–7.11).

```
TvarTypeSym = a symbol ending with a digit

A-list = Listof(Pair(TvarTypeSym, TvarTypeSym))
```


Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

```
equal-up-to-gensyms? : S-exp × S-exp → Bool
(define equal-up-to-gensyms?
  (lambda (sexp1 sexp2)
    (equal?
      (apply-subst-to-sexp (canonical-subst sexp1) sexp1)
      (apply-subst-to-sexp (canonical-subst sexp2) sexp2))))

canonical-subst : S-exp → A-list
(define canonical-subst
  (lambda (sexp)
    loop : S-exp × A-list → A-list
    (let loop ((sexp sexp) (table '()))
      (cond
        ((null? sexp) table)
        ((tvar-type-sym? sexp)
          (cond
           ((assq sexp table) table)
           (else
             (cons
               (cons sexp (ctr->ty (length table)))
               table))))
        ((pair? sexp)
         (loop (cdr sexp)
           (loop (car sexp) table)))
         (else table)))))
```

Figure 7.10: equal-up-to-gensyms?, part 1

```
tvar-type-sym? : Sym → Bool
(define tvar-type-sym?
  (lambda (sym)
    (and (symbol? sym)
      (char-numeric? (car (reverse (symbol->list sym)))))))

symbol->list : Sym → List
(define symbol->list
  (lambda (x)
    (string->list (symbol->string x))))

apply-subst-to-sexp : A-list × S-exp → S-exp
(define apply-subst-to-sexp
  (lambda (subst sexp)
    (cond
      ((null? sexp) sexp)
      ((tvar-type-sym? sexp)
       (cdr (assq sexp subst)))
      ((pair? sexp)
       (cons
         (apply-subst-to-sexp subst (car sexp))
         (apply-subst-to-sexp subst (cdr sexp))))
      (else sexp))))

ctr->ty : N → Sym
(define ctr->ty
  (lambda (n)
    (string->symbol
      (string-append "tvar" (number->string n)))))
```

Figure 7.11: equal-up-to-gensyms?, part 2

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 figure 4.1.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

$$\texttt{letrec}\ t_{proc\text{-}result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}\ :$$
$$t_p = t_{var} \rightarrow t_{e_{proc\text{-}body}}$$
$$t_{e_{letrec\text{-}body}} = t_{\texttt{letrec}\ t_{proc\text{-}result}\ p\ (var\ :\ t_{var}) = e_{proc\text{-}body}\ \texttt{in}\ e_{letrec\text{-}body}}$$

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

Figure 7.9: `type-of` for INFERRED, part 4

Exercise 7.23 [**] Extend the inferencer to handle pair types, as in exercise 7.8.

Exercise 7.24 [**] Extend the inferencer to handle multiple let declarations, multiargument procedures, and multiple letrec declarations.

Exercise 7.25 [**] Extend the inferencer to handle list types, as in exercise 7.9. Modify the language to use the production

```
Expression ::= emptylist
```

instead of

```
Expression ::= emptylist_Type
```

As a hint, consider creating a type variable in place of the missing _*t*.

Exercise 7.26 [**] Extend the inferencer to handle EXPLICIT-REFS, as in exercise 7.10.

Exercise 7.27 [**] Rewrite the inferencer so that it works in two phases. In the first phase it should generate a set of equations, and in the second phase, it should repeatedly call unify to solve them.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited

Exercise 7.28 [**] Our inferencer is very useful, but it is not powerful enough to allow the programmer to define procedures that are polymorphic, like the polymorphic primitives pair or cons, which can be used at many types. For example, our inferencer would reject the program

```
let f = proc (x : ?) x
in if (f zero?(0))
   then (f 11)
   else (f 22)
```

even though its execution is safe, because f is used both at type (bool→ bool) and at type (int→ int). Since the inferencer of this section is allowed to find at most one type for f, it will reject this program.

For a more realistic example, one would like to write programs like

```
let
 ? map (f : ?) =
    letrec
     ? foo (x : ?) = if null?(x)
                     then emptylist
                     else cons((f car(x)),
                               ((foo f) cdr(x)))
    in foo
in letrec
    ? even (y : ?) = if zero?(y)
                     then zero?(0)
                     else if zero?(-(y,1))
                          then zero?(1)
                          else (even -(y,2))
   in pair(((map proc(x : int)-(x,1))
            cons(3,cons(5,emptylist))),
           ((map even)
            cons(3,cons(5,emptylist))))
```

This expression uses map twice, once producing a list of ints and once producing a list of bools. Therefore it needs two different types for the two uses. Since the inferencer of this section will find at most one type for map, it will detect the clash between int and bool and reject the program.

One way to avoid this problem is to allow polymorphic values to be introduced only by let, and then to treat (let-exp *var* e_1 e_2) differently from (call-exp (proc-exp *var* e_2) e_1) for type-checking purposes.

Add polymorphic bindings to the inferencer by treating (let-exp *var* e_1 e_2) like the expression obtained by substituting e_1 for each free occurrence of *var* in e_2. Then, from the point of view of the inferencer, there are many different copies of e_1 in the body of the let, so they can have different types, and the programs above will be accepted.

Exercise 7.29 [***] The type inference algorithm suggested in the preceding exercise will analyze e_1 many times, once for each of its occurrences in e_2. Implement Milner's Algorithm W, which analyzes e_1 only once.

Exercise 7.30 [***] The interaction between polymorphism and effects is subtle. Consider a program starting

```
let p = newref(proc (x : ?) x)
in ...
```

1. Finish this program to produce a program that passes the polymorphic inferencer, but whose evaluation is not safe according to the definition at the beginning of the chapter.
2. Avoid this difficulty by restricting the right-hand side of a let to have no effect on the store. This is called the *value restriction*.

Reprinted for F7ZDY/ryho8890, Library
The MIT Press, Daniel P. Friedman and Mitchell Wand (c) 2008, Copying Prohibited