Type Checking and Inference

Note: this is a draft of a chapter destined for the second edition of *Essentials of Programming Languages,* by Friedman, Wand, and Haynes.

The data that programs manipulate come in many different types: integers, characters, procedures, lists, and so on. Part of the job of a programming language is to prevent inappropriate operations from being performed on such data. For example, it would be inappropriate for two characters to be multiplied, or for an integer to be applied to an argument. An attempt to perform such an operation is called a *type error*.

The progamming language fragments introduced in this book, like Scheme, rely on dynamic type checking to detect type errors. For example, the procedure `apply-proc` in our interpreters checks to see that its first argument is a valid representation of a procedure before applying it. In `apply-primop` no explicit tests are performed to insure that arguments to numeric primitives are numbers. We rely on the implementation of Scheme's arithmetic procedures to ensure that arithmetic is performed on numbers and not on procedures or characters.

Dynamic type checking requires that a tag be associated with each datum, indicating the primitive data type to which it belongs. This strategy imposes minimal restriction on program structure. It is particularly well-suited to interactive programming environments, such as those typically associated with Lisp-based languages. A price is paid, however, for the flexibility of dynamic type checking. Type errors are only reported when a program runs, so if a program is inadequately tested a type error might not be detected until the program is used in production. Also, the maintenance and checking of type tags may adversely affect run-time efficiency.

Static type checking is an alternative to dynamic type checking. If a language is carefully designed, it may be possible to analyze the text of a program, without knowledge of run-time data, to predict whether type errors can occur. This analysis is performed by a *type checker,* typically as part of the compilation process, though a type checker may also be used independently or in conjunction with an interpreter (as in this chapter). A program that is accepted by the type checker is said to be *well-typed*; otherwise, it is *ill-typed* and should be rejected.

Static type checking not only allows potential type errors to be detected at compile time, but it also eliminates the need for type tags on data (except as may be required for garbage-collection purposes), and may in other ways assist in the generation of more efficient compiled code. Furthermore, the types of program elements provide valuable documentation, whether they are provided as part of the program or displayed by the type checker.

A type discipline is said to be *sound* or *type-safe* if every well-typed program is guaranteed to execute without type errors; it is *unsound* if execution of a well-typed program may nevertheless cause type errors. Even an unsound type discipline may be useful if it prevents some type errors; C and Eiffel are examples of languages with useful but unsound type disciplines. The type disciplines considered in this chapter will be sound.

It is not always possible to determine whether a program will cause a type error. Consider the program

```
define p = proc (x) if strange(x) then 1 else 1(1)
```

This program causes a dynamic type error whenever `strange(x)` returns a false value. It may not always be possible, however, to determine whether `strange(x)` can return a false value, even if it were possible to examine the code of `strange`. (Consider the Halting Problem.) This situation is analogous to the problem of determining whether a program has iterative behavior (section 8.3).

Thus the advantages of a static type discipline are somewhat offset by the "unnecessary" restrictions that it imposes. More elaborate type disciplines may impose fewer restrictions, but they complicate type checking and make it more difficult for a programmer to understand why an ill-typed program was rejected by the type checker.

Traditional statically-typed programming languages, such as C and Pascal, require that the programmer specify the types of variables at the point at which they are declared. This greatly simplifies the type checker's job. In section 13.1 we study a type checker of this sort for a simple type discipline constructed from a few primitive types and the operator `->`, expressing functional types. In section 13.2 we introduce additional type construction operators for expressing sum and product types, and sketch how the type checkers of this chapter can be extended for these types.

In languages where type expressions rarely become large, the burden of declaring the types of variables is small, and may be more than offset by the value of such declarations as documentation. In other languages, especially those that encourage the use of functional programming techniques, type expressions are frequently quite large. Mandatory type declarations can then be a burden. Not surprisingly, this has made type disciplines that do not require type declarations increasingly popular. These disciplines require more sophisticated type checkers employing *type inference* algorithms. (Though any type checker must infer the types of subexpressions, we reserve the term type inference for cases in which the types associated with variables must be inferred.) We examine a type inference algorithm in section 13.3.

Some values may be regarded as having more than one type. For example, a `length` procedure might take a list of integers and return an integer, or it might take a list of boolean values and return an integer, and it could be used in both ways (at different applications) in the same program. Such a value is said to be *polymorphic*. In section 13.4, we study a widely-used polymorphic type inference algorithm.

13.1 Type Checking

In this section we present a type checking algorithm for a simple language. The expressions of this language are composed of forms used previously with two additions: the boolean literals `true` and `false`, and a new `assert` form. See figure 13.1.1 for the concrete and abstract syntax.

The `assert` form declares that an expression is of a given type. It may be used for documentation or debugging purposes to declare the type of any expression, and the type checker will verify that the given type is correct. An assert expression has no effect at run-time other than evaluation of the expression it contains, yielding the value of the assert expression.

For the type checker in this section, we require that the `assert` form be used in enough places so that the type of every variable will be known:

1. an `assert` form must be used around every ⟨proc⟩ form (that is, every ⟨proc⟩ form must appear as the ⟨exp⟩ component of an assert form), and
2. every `letrec` declaration expression (an ⟨exp⟩ component of a ⟨decl⟩ belonging to a ⟨decls⟩ list of a `letrec`) must be an `assert` form.

⟨exp⟩	::=	⟨integer-literal⟩ | `true` | `false`	`lit (datum)`
	|	⟨varref⟩	`varref (var)`
	|	⟨operator⟩ ⟨operands⟩	`app (rator rands)`
	|	`if` ⟨exp⟩ `then` ⟨exp⟩ `else` ⟨exp⟩	`if (test-exp then-exp else-exp)`
	|	`proc` ⟨varlist⟩ ⟨exp⟩	`proc (formals body)`
	|	`let` ⟨decls⟩ `in` ⟨exp⟩	`let (decls body)`
	|	`letrec` ⟨decls⟩ `in` ⟨exp⟩	`letrec (decls body)`
	|	`assert` ⟨type⟩ : ⟨exp⟩	`assert (type exp)`
⟨type-list⟩	::=	`()` | `(`⟨type⟩ {`,` ⟨type⟩}* `)`	
⟨type⟩	::=	⟨prim-type⟩ | ⟨arrow-type⟩	`tcons (name types)`
⟨prim-type⟩	::=	`int` | `bool`	
⟨arrow-type⟩	::=	`(->` ⟨type-list⟩ ⟨type⟩`)`	

Figure 13.1.1 Syntax of simple typed language

```
proc (v1,...,vn) e
 |    |  ...  |  |
(->  (t1,...,tn) t)
```

Figure 13.1.2 Procedure and arrow type syntax

As a consequence, whenever the simple type checking algorithm encounters a variable, its type will already be known.

We further require that a `letrec` declaration expression be a procedure, or rather at present an assert statement immediately containing a procedure. This is the simplest way of guaranteeing that during evaluation of a `letrec` declaration expression, no `letrec` bound variable will be referenced before it is bound.

These constraints are enforced by the type checker rather than the parser. We do not build these constraints into the grammar because in section 13.3 `assert` forms are not required, but the language otherwise remains unchanged.

A type expression, ⟨type⟩, is either one of the primitive types, `int` (integer) or `bool` (boolean), or an arrow type, indicating the type of a procedure. In an arrow type, the types of the arguments to the procedure are listed in parentheses and are called the *domain* types of the procedure. The domain

Figure 13.1.3 Typed program example

```
let decrement = assert (-> (int) int) : proc (n) +(n, -1);
    compose = assert (-> ((-> (int) bool), (-> (int) int))
                         (-> (int) bool)) :
                proc (f, g)
                  assert (-> (int) bool) :
                    proc (n) f(g(n))
in let isone = compose(zero, decrement)
   in isone(2)
```

types are followed by the type of the result, which is called the *range* type of the procedure. Figure 13.1.2 illustrates the parallels between the syntax of a procedure and the syntax of its type. An arrow type is an example of a *compound type,* since it contains type subexpressions.

The example of figure 13.1.3 illustrates the use of type expressions and the assert form. We assume that the primitive procedures `+` and `zero` are provided by the initial environment, with types `(-> (int, int) int)` and `(-> (int) bool)`, respectively.

We represent all the types in our languages as *type constructions.* A type construction consists of a symbol naming the form of the type, and a list of type subexpressions. We use the abstract syntax

```
(define-record tcons (name types))
```

For a primitive type, the `name` field contains the symbol `int` or `bool` and the `types` field contains the empty list. For arrow types, the `name` field contains the symbol `->`, and the `types` field contains a list whose car is the range type and whose cdr is a list of the domain types. The range type is at the head of the list simply for convenience; there is only one range type, but there may be any number of domain types.

Before studying the type-checking algorithm, we introduce a few auxiliary definitions for recognizing and manipulating arrow types and generating primitive types. See figure 13.1.4. The `(tcons? type)` call in the definition of *proc-type?* is included because in upcoming type checkers a type will not always be a `tcons` record.

For the type checker of this section, two types match if they are structurally identical (they print the same way). See figure 13.1.5 for the definition of *match-types*, as well as *match-types-pairwise*, which is used to match lists of types. The auxiliary procedure *andmap2* accepts a pred-

Figure 13.1.4 Auxiliary procedures for manipulating types

```
(define domains (compose cdr tcons->types))
(define range (compose car tcons->types))

(define integer-type (make-tcons 'int '()))
(define boolean-type (make-tcons 'bool '()))

(define type-of-datum
  (lambda (datum)
    (if (integer? datum) integer-type boolean-type)))

(define proc-type?
  (lambda (type)
    (and (tcons? type) (eq? '-> (tcons->name type)))))

(define make-proc-type
  (lambda (domain-types range-type)
    (make-tcons '-> (cons range-type domain-types))))
```

icate of two arguments followed by two lists of equal length, and returns true only if the predicate always returns true when passed arguments obtained from the same positions in each of the lists. Calls to *match-types* and *match-types-pairwise* will not return if a pair of types fails to match; the entire type-checking computation is aborted with an error message when the first type error is found.

Type errors are reported by calling the procedure *type-error*; see figure 13.1.6. Its first argument is a string that identifies the nature of the error. Subsequent arguments are types that are known to be involved in the error. Following an identifying message, *type-error* prints these types after first unparsing them. (Definition of the procedure *unparse-type* is left as an exercise.) Finally, the type-checking computation is aborted by calling the procedure *error*.

In much the same way that interpreters use environments to associate variables with values, type checkers use *type environments* to associate variables with the types of the values to which they can be bound. We assume that a type environment ADT supplying the procedures *init-tenv*, *extend-tenv*, and *apply-tenv* has been defined, and that the initial type environment contains the types of the variables that will be bound in the initial environment of the interpreter. Thus, if reference is made to an unbound variable, the

Figure 13.1.5 The *match-types* procedures

```
(define match-types
  (letrec
    ((same-type?
       (lambda (type1 type2)
         (let ((types1 (tcons->types type1))
               (types2 (tcons->types type2)))
           (and (eq? (tcons->name type1) (tcons->name type2))
             (= (length types1) (length types2))
             (andmap2 same-type? types1 types2))))))
    (lambda (type1 type2)
      (if (not (same-type? type1 type2))
        (type-error "Incompatible types:" type1 type2)))))

(define match-types-pairwise
  (lambda (types1 types2)
    (for-each match-types types1 types2)))

(define andmap2
  (lambda (predicate list1 list2)
    (or (null? list1)
      (and (predicate (car list1) (car list2))
        (andmap2 predicate (cdr list1) (cdr list2))))))
```

Figure 13.1.6 Type error procedure

```
(define type-error
  (lambda ls
    (display "Type error: ")
    (display (car ls))
    (newline)
    (for-each
      (lambda (type) (displayln (unparse-type type)))
      (cdr ls))
    (error)))
```

error will be caught by the type checker, prior to run time.

The main driver for the simple type checker is *type-of-exp*; see figure 13.1.7. The type checkers of section 13.3 and section 13.4 will employ

Figure 13.1.7 Driver for type checkers

```
(define type-of-exp
  (lambda (exp tenv)
    (trace-entry exp)
    (let ((answer
            (variant-case exp
              (lit (datum) (type-of-datum datum))
              (varref (var) (apply-tenv tenv var))
              (if (test-exp then-exp else-exp)
                (let ((test-exp-type (type-of-exp test-exp tenv))
                      (then-exp-type (type-of-exp then-exp tenv))
                      (else-exp-type (type-of-exp else-exp tenv)))
                  (match-types test-exp-type boolean-type)
                  (match-types then-exp-type else-exp-type)
                  then-exp-type))
              (app (rator rands)
                (type-of-app
                  (type-of-exp rator tenv)
                  (types-of-rands rands tenv)))
              (let (decls body)
                (type-of-exp body
                  (extend-tenv
                    (map decl->var decls)
                    (types-of-let-rands decls tenv)
                    tenv)))
              (letrec (decls body)
                (for-each (compose check-letrec-decl-exp decl->exp)
                  decls)
                (type-of-exp body (letrec-tenv decls tenv)))
              (assert (type exp) (type-of-assert type exp tenv))
              (proc (formals body) (type-of-proc formals body tenv))
              (else (error "Invalid abstract syntax:" exp)))))
      (trace-exit answer)
      answer)))

(define types-of-rands
  (lambda (rands tenv)
    (map (lambda (rand) (type-of-exp rand tenv))
      rands)))
```

Figure 13.1.8 Auxiliaries for tracing type checkers

```
(define trace-depth-counter 0)

(define trace-type-of-exp #t)

(define trace-entry
  (lambda (exp)
    (if trace-type-of-exp
      (begin
        (set! trace-depth-counter (+ trace-depth-counter 1))
        (displayln trace-depth-counter " Entering with "
          (unparse exp))))))

(define trace-exit
  (lambda (type)
    (if trace-type-of-exp
      (begin
        (displayln trace-depth-counter " Leaving with "
          (unparse-type type))
        (set! trace-depth-counter (- trace-depth-counter 1))))))
```

the same driver with different auxiliary procedures. The expressions passed to *type-of-exp* and the types returned are traced using the auxiliary procedures in figure 13.1.8.

Like *eval-exp*, *type-of-exp* is *syntax directed*; that is, it dispatches (via `variant-case`) on the form of the expression, and calls itself recursively on subexpressions. Since recursive calls by the type checker are always on proper subexpressions of the input expression, and the auxiliary procedures always terminate, type checking is guaranteed to terminate. (Why is this not the case for *eval-exp*?)

The type of a literal expression is obtained by calling *type-of-datum*, and the type of a variable reference is obtained from the type environment. For an `if` expression, we check that the type of the test expression is boolean and that the types of the `then` and `else` expressions match. (The values of the `then` and `else` expressions must be the same since they are both returned as the value of the `if` expression.) The *match-types* procedure described above aborts the computation if either of these checks fails. Otherwise, the type of the `then` (or `else`) expression is returned as the type of the `if` expression.

For an application, the types of the operator and operand expressions are computed and passed to the auxiliary procedure `type-of-app`, which checks that the operand types match the domain types of the operator and returns the range type of the operator type; see figure 13.1.9. If the operator does not have a procedural type, the computation is aborted with an error message.

The type of a `let` or `letrec` expression is the type of its body, computed with a type environment obtained by extending the current type environment by associating each declared variable with the type of its corresponding declaration expression. For a `let` expression, a list containing the types of the variable declaration expressions is returned by `types-of-let-rands`. Each of these types is obtained by calling `type-of-exp` with the type environment of the whole `let` expression.

For a `letrec` expression, the procedure `check-letrec-decl-exp` is first applied to all the declaration expressions to verify that they are `assert` expressions that immediately contain procedure expressions. The procedure `letrec-tenv` is then called to construct the new environment in which the body is to be checked. Since each declaration expression is an `assert` expression, the new type environment is obtained by extracting the type of each declaration expression from its type subexpression. The `for-each` expression in the body of the procedure `letrec-tenv` then checks that each declaration expression is well-typed in the new environment; if any of the declaration expressions is ill-typed, an error is reported and the checker stops. Finally, the new environment is returned, and the body of the `letrec` is checked in the new environment.

Assert expressions are checked by `type-of-assert`. If the subexpression is a procedure, we first check that the asserted type is a procedure type. Then we check that the type of the procedure's body matches the range type of the asserted type. The body is checked in a type environment obtained by extending the current environment with associations of the procedure's formal parameters with the corresponding domain types of the asserted type. If the asserted type is not procedural, we verify that the type of the subexpression (in the current type environment) matches the asserted type. In both cases, we return the asserted type if no error is found.

Procedure expressions passed directly to `type-of-exp` are checked by `type-of-proc`. If a procedure expression is passed directly to `type-of-exp` an error is reported, since `proc` expressions can only appear immediately within assert expressions.

• *Exercise 13.1.1*

For each of the following expressions in the language of this section, indicate its type if it is well-typed or the sort of error message that the type checker

Figure 13.1.9 Auxiliary procedures for a simple type checker

```
(define type-of-app
  (lambda (rator-type rand-types)
    (if (proc-type? rator-type)
      (match-types-pairwise (domains rator-type) rand-types)
      (type-error "Not a procedure type:" rator-type))
    (range rator-type)))

(define types-of-let-rands
  (lambda (decls tenv) (types-of-rands (map decl->exp decls) tenv)))

(define check-letrec-decl-exp
  (lambda (exp)
    (if (not (and (assert? exp) (proc? (assert->exp exp))))
      (error "Invalid declaration expression:" exp))))

(define letrec-tenv
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (assert-exps (map decl->exp decls)))
      (let ((var-types (map assert->type assert-exps)))
        (let ((new-tenv (extend-tenv vars var-types tenv)))
          (for-each
            (lambda (assert-exp) (type-of-exp assert-exp new-tenv))
            assert-exps)
          new-tenv)))))

(define type-of-assert
  (lambda (assert-type exp tenv)
    (variant-case exp
      (proc (formals body)
        (if (proc-type? assert-type)
          (if (= (length formals) (length (domains assert-type)))
            (match-types (range assert-type)
              (type-of-exp body (extend-tenv formals (domains assert-type) tenv)))
            (error "Domain does not match formals" formals assert-type))
          (type-error "Assert type is not a procedure type:" assert-type)))
      (else (match-types (type-of-exp exp tenv) assert-type)))
    assert-type))

(define type-of-proc
  (lambda (formals body tenv) (error "Procedure not inside an assert:" formals body)))
```

will issue if it is ill-typed. Assume + has type `(-> (int, int) int)`.

1. `if true then 3 else +(2, +(1, 5))`
2. `if +(1, 2) then 3 else 4`
3. `(proc (x) +(x, 3))(5)`
4. `let f = assert (-> (int) int) : proc (x) +(x, 3) in f(5)`
5. `letrec x = 3 in +(x, 5)`

□

• *Exercise 13.1.2*
Implement the procedure *unparse-type*, which takes a type in the abstract syntax of figure 13.1.1 and returns a Scheme list structure suitable for printing. For example, *unparse-type* might be tested with the type of the *compose* procedure of figure 13.1.3 as follows:

```
> (unparse-type
    (make-proc-type
      (list (make-proc-type (list integer-type) boolean-type)
        (make-proc-type (list integer-type) integer-type))
      (make-proc-type (list integer-type) boolean-type)))
(-> ((-> (int) bool) (-> (int) int)) (-> (int) bool))
```

The Scheme list structure returned by *unparse-type* prints with the same concrete syntax as that of figure 13.1.3, except that multiple domain types are not separated by commas. □

• *Exercise 13.1.3*
Implement the type checker of this section. You will have to implement *parse-type*, as well as *parse-exp*. Use the procedure *unparse-type* of exercise 13.1.2 to view the type checker output. An initial type environment providing the types of common primitive procedures is also required. □

• *Exercise 13.1.4*
Implement an interpreter for the language of this section along with a read-eval-print loop. The type of each top-level expression should be printed following its value. Use *unparse-type* of exercise 13.1.2. For example,

```
--> +(1,2)
3 : int
--> zero(1)
false : bool
```

To obtain the value of an assert expression, simply evaluate its subexpression. If the type checker detects a type error in the top-level expression, then the expression should not be evaluated, but control should stay in the read-eval-print loop. See exercise 9.3.? [[to be added in the second edition]]. □

- *Exercise 13.1.5*

Extend the language of exercise 13.1.4 by adding a `define` form, which may be used to make non-recursive top-level declarations. Also add a `definerec` form, which may be used to make mutually-recursive top-level declarations that are restricted in the same manner as `letrec` declarations. The syntax for `define` declarations previously introduced in exercise 5.5.5 may be used. Invent appropriate concrete and abstract syntax for `definerec`. Print new variable/type associations added to the top-level type environment; for example,

```
--> define add1 = assert (-> (int) int) :
                    proc (x) +(x,1)
add1 :: (-> (int) int)
--> add1(3)
4 : int
```

(A double, rather than single, colon is used when indicating a variable/type association because the variable name is not a value.) □

- *Exercise 13.1.6*

Extend the language of exercise 13.1.5 to admit definitions of the form

⟨form⟩ ::= `definetypeabbreviation` ⟨type-name⟩ ⟨type⟩

and allow a *type* to be simply a *type-name* (a symbol other than `int` or `bool`). The effect of such a type abbreviation is that whenever a type name appears as a type, it is interpreted as if the corresponding type were substituted in its place. Maintain type abbreviations in printed types whenever practical; for example,

```
--> definetypeabbreviation intproc (-> (int) int)
--> define compose = assert (-> (intproc, intproc) intproc) :
                       proc (f, g) assert intproc proc (x) f(g(x))
compose :: (-> (intproc, intproc) intproc)
```

You will need to introduce a new abstract syntax record type for type names. Type abbreviations must be defined before they are used. This prohibits forward references to type abbreviations and recursive type abbreviations, which should be reported as errors. □

13.2 Additional Type Operators

For simplicity, the only compound types in our language have been procedural types. Practical type systems typically support several other compound types and may allow the user to define new ones. In addition to the function type construction arrow, $\rightarrow$, the most basic compound type constructors are the *product,* $\times$, and *sum,* $+$.

Products, also called *cross products* or *Cartesian products*, are used to type data structures with multiple components. For example, the type of a pair (cons cell) might be represented as the product of the types of its left and right components. Heterogeneous data structures are called *records* or *structures* if their components are referenced by a name, and *tuples* if their components are referenced by a numerical index. An *array* is another form of compound data structure whose components are referenced by numerical index, but arrays are homogeneous (all their components are of the same type).

Associated with each product type t is a *constructor* and a set of *selectors,* one for each component of the product type. The constructor takes a value for each component and returns a corresponding aggregate data value of type t. Selectors take a value of type t and return the value of their corresponding component. In the case of Scheme pairs, the constructor is `cons` and the selectors are `car` and `cdr`.

Products are often expressed in n-ary, rather than binary form. For example, assuming expressions $e_1, \ldots, e_n$ yield values of type $t_1, \ldots, t_n$, respectively, a tuple expression of the form $\texttt{<}e_1\texttt{,} \ldots\texttt{,}\ e_n\texttt{>}$ would have type $t_1 \times \ldots \times t_n$.

An array of three elements of type t might be typed by the product $t \times t \times t$, but this approach would not work well for an array of 100 elements. It is customary, instead, to introduce a new type constructor, *array*, with a single subtype indicating the element type and in many cases no provision for indicating the size of the array. Thus all arrays with elements of type t have the same type, $array(t)$.

Sums, also called *disjoint unions* or *discriminated unions*, are used to type data that may be represented by information of more than one type. We have already encountered sums in several contexts. The $+$ operator was used in equations specifying denoted and expressed values, which are in effect the types of the data manipulated by the interpreters. A BNF grammar may be viewed as specifying the type of each syntactic category, with the alternation operator ("$\mid$") expressing sums. In a `variant-case` expression, each record name identifies a variant of a sum type. Finally, lists have sum types, since they may be represented by either of two variants: the empty list, or a tuple containing a head and a tail. Sums are sometimes called *unions,* but should not be confused with simple set-theoretic unions, which may not be disjoint. Each of the alternative types that make up a sum is called a *variant* of the sum.

The procedures for manipulating sum types are different from those for product types. If t is a sum type

$$t = t_1 + \ldots + t_n$$

then for each variant t_i one has the following procedures:

- an *injection* inj_i, of type $t_i \to t$,
- a *projection* proj_i, of type $t \to t_i$, and
- a *predicate* pred_i, of type $t \to \texttt{bool}$.

For a value x of type t, $pred_i(x)$ is true if and only if x has been obtained by injection into the ith variant of t. The projection and injection elements of a given variant are semi-inverses; thus if y has type t_i, then $y = \mathrm{proj}_i(\mathrm{inj}_i(y))$. If a value has been obtained by injection into the ith variant of a sum, however, it is an error to attempt projection of it to any but the i-th variant; thus if $i \neq j$, $\mathrm{proj}_j(\mathrm{inj}_i(y))$ will result in a runtime error.

In many programming languages, an element of sum type $t = t_1 + \ldots + t_n$ is represented as a tuple of two components. The first component is a tag indicating to which sum variant the value belongs, say i, while the second component is the actual value of type t_i. The sum injection operation may be thought of as adding a tag, while projection strips off the tag, and the predicate pred_i tests whether the tag is i. The C language does not have sum types, but using this approach they may be simulated using a product type and C's union type. For example, the declaration

```
typedef struct {
    int tag;
    union {
        type1 variant1;
        type2 variant2;
        } u;
    } sumtype;
sumtype *x;
```

defines `sumtype` to be a sum of `type1` and `type2` and `x` to be a pointer to a structure of type `sumtype`. Then we could write `x->u.variant1` to project `x` on `type1` or `x->u.variant2` to project on `type2`. Pascal uses a similar mechanism, called *variant records*. Neither of these mechanisms is sound, because they allow a value injected from t_i to be manipulated as if it were from t_j, and tags must be inserted manually.

In a number of programming languages with sound sum types, the sum types must be declared, and their variants are assumed to be products. Typically such a sum type declaration will name each of the variants of the sum, each component of all of the products associated with a sum variant, and the sum type itself. A single operation may be used to both construct a product and inject it into the sum, while other operations may both project to a value of product type and then select a component of the product. For example, a definition like

```
definesumtype intlist
  emptyintlist (),
  intcons (car : int, cdr : intlist);
```

would create the following procedures with the indicated types:

```
emptyintlist     :  (-> () intlist)
emptyintlistpred :  (-> (intlist) bool)
intcons          :  (-> (int, intlist) intlist)
intconspred      :  (-> (intlist) bool)
intconscar       :  (-> (intlist) int)
intconscdr       :  (-> (intlist) intlist)
```

We have used sets of `define-record` declarations in this way to effectively define sums of product types. The record names identify the sum variants, and record field names identify the product components, but this mechanism does not name the sum type itself. A record with no fields may be used when a sum variant only conveys the identity of its component.

In general, a type discipline based on the projection functions proj_i is not sound, since an application of proj_i to some value not injected from the i-th variant is an error. This is the situation in languages like C or Pascal, where performing an operation equivalent to $\mathrm{proj}_i(\mathrm{inj}_j(x))$ may lead to unpredictable results. Soundness can be guaranteed by including a call to pred_i inside proj_i, so that an error of the form $\mathrm{proj}_i(\mathrm{inj}_j(x))$ will be detected at execution time, and an appropriate error message generated. Another approach to soundness is to replace the projection functions with something more like `variant-case`.

• *Exercise 13.2.1*

Extend the typed language of the last section to include tuple expressions and product type expressions. Also include expressions of the form `select` *nat* `of` *exp* that return the tuple component indicated by the given natural number. Use zero-based indexing. For example,

```
--> define tup < +(1,2), zero(3) >
tup :: product(int, bool)
--> select 1 of tup
false : bool
```

□

∘ *Exercise 13.2.2*
Extend the typed language of exercise 13.1.5 to include sum type declarations like those discussed above. A possible syntax is

$$
\begin{aligned}
\langle\text{form}\rangle &::= \texttt{definesumtype}\ \langle\text{type-name}\rangle\ \langle\text{variant}\rangle\ \{\texttt{,}\langle\text{variant}\rangle\}^* \\
\langle\text{variant}\rangle &::= \langle\text{variant-name}\rangle\ \langle\text{field-list}\rangle \\
\langle\text{field-list}\rangle &::= \texttt{()}\ |\ \texttt{(}\langle\text{field}\rangle\ \{\texttt{,}\langle\text{field}\rangle\}^*\texttt{)} \\
\langle\text{field}\rangle &::= \langle\text{field-name}\rangle\texttt{:}\langle\text{type}\rangle
\end{aligned}
$$

The effect of this declaration is to define a set of injection, projection, and predicate procedures similar to those that would be defined by an equivalent set of `define-record` definitions. Use the naming conventions suggested by the `intlist` example above. Recursive sum types, like the one in the `intlist` example, should be allowed. □

13.3 Type Inference

In this section we modify our type checking algorithm so that it is capable of determining the types associated with variables. Assert expressions then need not be associated with procedure and `letrec` expressions.

To see how this might be accomplished, consider the expression

```
proc (p, x) p(x, +(x, 1))
```

By examining this code, we can deduce some facts about its type, without needing any `assert` expressions. First, `p` is applied to two arguments, so its type must be of the form

$$\texttt{(-> (}t_1\texttt{, }t_2\texttt{) }t_3\texttt{)}$$

for some types t_1, t_2, and t_3 that we have yet to determine. The second argument to `p` must be an `int`, so t_2 must be `int`. Therefore the type of `p` must be

$$\texttt{(-> (}t_1\texttt{, int) }t_3\texttt{)}$$

Furthermore, the type of `x` must be `int`, since `x` is an argument to `+`. Thus t_1 must also be `int`. The result of the application of p is the result of the procedure body, so the result of the procedure must be of type t_3. Putting all this together, we deduce that the type of the entire expression must be of the form

$$\texttt{(-> ((-> (int, int) }t_3\texttt{) int) }t_3\texttt{)}$$

for some type t_3.

In general, if e is an application e_0`(`e_1`, ... , `e_n`)`, where $t_0, \ldots, t_n$, and t are the types of $e_0, \ldots, e_n$, and e, then e_0 must accept n arguments of types $t_1, \ldots, t_n$, and it must return a value of type t. Expressed as an equation, this becomes

$$t_0 = \texttt{(-> (}t_1, \ldots, t_n\texttt{) }t\texttt{)}$$

We can think of this as a constraint on the possible values of $t_0, \ldots, t_n$, and t, or equivalently as an equation to be solved for the *type variables* $t_0, \ldots, t_n$, and t.

The same process of equation solving can be used to detect potential type errors, because a program with a potential type error will yield a set of equations that has no solution. For example, consider the expression

```
proc (p, x) p(x(1), +(x, 1))
```

In the subexpression `x(1)`, `x` is applied to an integer, so the type of `x` must be of the form

$$\texttt{(-> (int) }t_1\texttt{)}$$

for some type t_1. However, `x` also appears as the first argument to `+`, and thus it must be of type `int`. From these two inferences we conclude

$$\texttt{(-> (int) }t_1\texttt{)} = \texttt{int}$$

but this equation is not solved by any value of t_1, since an arrow type is not the same as a primitive type.

In this manner all the type constraints that the type discipline implies for a given program can be expressed as a set of type equations. These equations can then be solved using a technique called *unification*. This will either yield the type of the entire program and of every expression and variable in it, or we will discover that the equations have no solution, indicating that the program is ill-typed. We call this process *type inference*, rather than type checking, since it infers a type for each phrase in the program.

To accomplish this, we modify our previous type checker in two essential ways. First, we expand our type expressions to include type variables. Type variables will be used as place holders for portions of a type that are not yet known. Types that may contain type variables are called *polytypes,* while types that do not are called *monotypes.* A polytype may be nothing more than a type variable. In general, a polytype can be thought of as a structure for a type, where type variables represent portions of the structure to be filled in later. In our terminology, all monotypes are also considered to be polytypes.

Initially a type variable is *unbound*, meaning that we have no information about it. Type variables are said to be *bound* when their associated type, or at least some part of its structure, becomes known.

The second major modification to our type checker is in the procedure *match-types*. In the simple type checker, a call to *match-types* had no side-effects unless the types were not the same, in which case a type error was reported. Now that types may contain type variables, the *match-types* procedure checks whether two types *could* represent the same types if unbound type variables were bound to the appropriate types, in which case *match-types* binds the type variables to appropriate types. This process, called unification, solves the type equations as type inference proceeds.

13.3.1 A Type Inference Algorithm

We first present the upper levels of the new type checker. The main driver is still *type-of-exp*, defined in figure 13.1.7, but we now employ the auxiliary procedures of figure 13.3.1. The procedure *unify* performs unification, while the procedures *fresh-var-tvar* and *fresh-app-tvar* return new, initially unbound, type variables associated with variable binding occurrences and applications, respectively. These procedures will be defined in section 13.3.2.

The procedure *type-of-app* attempts to solve the equation t_0 = `(->` ($t_1, \ldots, t_n$) t`)` for each application. It does this by checking to see that the type of the operator, t_0, unifies with a newly constructed procedure type whose domain types, $t_1, \ldots, t_n$, are the types of the operands and whose range type is represented by a new type variable, t. The type variable t is returned; its binding will be the polytype that describes as much of the structure of the type of the expression as we have been able to deduce so far.

Similarly, for a `let` expression `let` v_1 `=` e_1 ... v_n `=` e_n `in` e_0, the procedure *types-of-let-rands* creates a type variable, t_{v_i}, for each bound variable v_i, and then solves the equations

$$\begin{aligned} t_{v_1} &= t_{e_1}, \\ &\vdots \\ t_{v_n} &= t_{e_n} \end{aligned}$$

where t_{e_1}, ..., t_{e_n} are the types of the expressions e_1, ..., e_n. A list of these binding types is returned as the value of *types-of-let-rands*, and then *type-of-exp* finds the type of the body e_0 in an extended type environment formed by associating each v_i with t_{v_i}. See figure 13.1.7. (The definition of *types-of-let-rands* in figure 13.1.9(rather than figure 13.3.1) could still be used, but type variable names would then be less meaningful.)

The procedure *letrec-tenv* also creates a fresh type variable for each program variable bound in the declaration. These type variables, as yet unbound, are then used to create the new type environment in which the declaration expression and `letrec` body are checked. This is similar to *types-of-let-rands*, except that the expressions e_1, ..., e_n are checked in the extended type environment rather than the original one. The procedure *check-letrec-decl-exp* is modified to make the use of the assert form in a declaration expression optional, while still requiring that evaluation of a declaration expression results in immediate creation of a closure. The procedure *type-of-assert* now simply checks that the asserted type is correct before it is returned; it is no longer necessary to treat procedure expressions as a special case. In the procedure *type-of-proc*, a fresh type variable is created for each formal parameter. The body of the procedure is then checked in a type environment extended by associating the formal parameters with the new type variables, yielding the range type. A new procedure type is then returned, containing the new type variables and the range type.

Last, the procedure *match-types* is redefined to call *unify*; however, *match-types-pairwise* need not be redefined since it only uses *match-types*.

13.3.2 Type Variables

We now turn to the representation of type variables and the unification process. The abstract syntax of types must be extended, allowing types to be represented either by type construction records or type variable records. The concrete syntax of types in programs is not extended, since types appearing in programs as part of `assert` expressions do not contain type variables.

Some type variables are associated with applications, and are named `t1`, `t2`, etc. Other type variables are associated with variable binding occurrences

Figure 13.3.1 Auxiliary procedures for type inference

```
(define type-of-app
  (lambda (rator-type rand-types)
    (let ((range-type (fresh-app-tvar)))
      (match-types rator-type (make-proc-type rand-types range-type))
      range-type)))

(define types-of-let-rands
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (exps (map decl->exp decls)))
      (let ((bound-variable-types (map fresh-var-tvar vars)))
        (match-types-pairwise bound-variable-types (types-of-rands exps tenv))
        bound-variable-types))))

(define letrec-tenv
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (exps (map decl->exp decls)))
      (let ((bound-variable-types (map fresh-var-tvar vars)))
        (let ((new-tenv (extend-tenv vars bound-variable-types tenv)))
          (match-types-pairwise bound-variable-types (types-of-rands exps new-tenv))
          new-tenv)))))

(define check-letrec-decl-exp
  (lambda (exp)
    (if (not (or (proc? exp) (and (assert? exp) (proc? (assert->exp exp)))))
      (error "Invalid declaration expressions:" exp))))
```

(in a `let`, `letrec`, or `proc` expression). The names of these type variables are composed of the letter `t` followed by the name of the associated variable. If there is more than one declaration with the same variable name, the type variables associated with all but the first declaration will have "^n" appended to their names, where n is a unique index number. Thus `tx`, `tx^1`, `tx^2`, ... are the type variable names associated with declarations of the variable `x`. Our type variable naming conventions aid in the interpretation of type error messages and traces, but have no other significance.

See figure 13.3.2 for an implementation of our type variable ADT. A type

Figure 13.3.1 Auxiliary procedures for type inference (continued)

```
(define type-of-assert
  (lambda (assert-type exp tenv)
    (match-types assert-type (type-of-exp exp tenv))
    assert-type))

(define type-of-proc
  (lambda (formals body tenv)
    (let ((domain-types (map fresh-var-tvar formals)))
      (let ((range-type (type-of-exp body (extend-tenv formals domain-types tenv))))
        (make-proc-type domain-types range-type)))))

(define match-types
  (lambda (type1 type2)
    (unify type1 type2)))
```

variable is represented as a record with three fields. The `name` field contains a symbol naming the type variable, and the `app?` field contains a boolean value indicating whether the type variable is associated with an application. These fields are not essential to the logic of the type inference algorithm, but they aid the generation of meaningful error messages and traces. To allow type variables to be bound, we store the binding of a type variable in a mutable cell in the `cell` field of a `tvar` record. Initially this cell contains a symbol indicating that the type variable has not yet been bound.

The procedure *fresh-tvar* constructs a new type variable whose name is indicated by a string. It is used only by *fresh-app-tvar* and *fresh-var-tvar*, which construct new type variables associated with applications and variable binding occurrences, respectively. The procedure *fresh-app-tvar* uses the global variable `app-index` to keep track of the number of application type variables created so far. Similarly, the procedure *fresh-var-tvar* uses the global variable `var-list`, which contains a list of the variables for which type variables have so far been created. The number of duplicates in this list is used to determine the index number that follows the ^ character in a type variable name, if one is required. The procedure *count-occurrences*, of section 2.2.8, counts the number of symbols in *var-list* associated with a given variable. The procedures *tvar-binding* and *tvar-unbound?* return type variable bindings and indicate whether a type variable is unbound, respectively, while *bind-tvar!* makes bindings. Again *displayln* is used to trace type variable binding operations.

Unifying types may involve checking if an unbound type variable, say u, is the same as a given type, say t. Such a check should succeed (unless t contains u—a possibility that will be considered in section 13.3.3), since u is a place holder that is so far unconstrained. The check, however, reveals that its place must be filled by t. This information is recorded by binding u to t. This forces the structure of u to be the same as the structure of t. If t is itself a polytype, there may be more information to be filled in later. If two unbound type variables are unified, one of the type variables (it does not matter which) is bound to the other.

Consider the behavior of the type checker on the following program.

```
proc (a, b, c)
  if x then a
  else if y then b
       else c
```

First, the initial type environment is extended to associate the formal parameters `a`, `b`, and `c` with fresh type variables whose name fields are `ta`, `tb`, and `tc`, respectively. (When we refer to a type variable by name, we mean the type variable whose name field contains the given name.) Since the types of `b` and `c` must be the same, the procedure *type-of-exp* calls *match-types* with `tb` and `tc`. The procedure *match-types* records the requirement that `tb` = `tc` by binding type variable `tb` to type variable `tc` (or vice-versa), as in figure 13.3.3(a), and the type variable `tb` is returned as the type of the inner `if` expression. To complete checking of the outer `if` expression, *match-types* is called with type variables `ta` and `tb`. To record the deduction that these type variables must also stand for the same type, `ta` is bound to `tb`, as illustrated in figure 13.3.3(b).

From this example we see that a type variable may be associated with a *binding path.* If we start at a type variable t and follow its binding path, we must eventually reach either an unbound type variable or some compound polytype (that is, a polytype that is not just a type variable). We call either of these the *end value* of t. We are always concerned with the end value of a type variable, rather than the type variable itself (unless it is unbound, in which case it is its own end value). All the type variables in the path are said to be members of the same *congruence class,* of which the end value serves as the unique representative. An unbound type variable represents its own congruence class.

If a type variable is some distance from the end of a binding path, it would be inefficient to run down the path every time the variable is encountered.

Figure 13.3.2 Type variable ADT

```
(define-record tvar (cell name app?))

(define app-index 0)

(define var-list '())

(define fresh-tvar
  (lambda (app? name)
    (make-tvar (make-cell '*unbound*) (string->symbol name) app?)))

(define fresh-app-tvar
  (lambda ()
    (set! app-index (+ 1 app-index))
    (fresh-tvar #t (string-append "t" (number->string app-index)))))

(define fresh-var-tvar
  (lambda (var)
    (let ((index (count-occurrences var var-list)))
      (set! var-list (cons var var-list))
      (fresh-tvar #f
        (string-append "t"
          (symbol->string var)
          (if (zero? index)
            ""
            (string-append "^" (number->string index))))))))

(define tvar-binding (compose cell-ref tvar->cell))

(define tvar-unbound?
  (lambda (tvar)
    (eq? (tvar-binding tvar) '*unbound*)))

(define trace-tvar-bindings #t)

(define bind-tvar!
  (lambda (tvar type)
    (if trace-tvar-bindings
      (displayln "  Binding tvar " (tvar->name tvar)
        " to " (unparse-type type)))
    (cell-set! (tvar->cell tvar) type)))
```

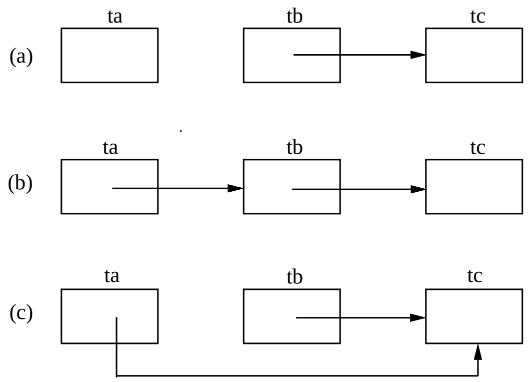

Figure 13.3.3 Growth of a type variable path

Thus whenever we encounter a bound type variable, we traverse its binding path to the end and change the contents of the variable's binding cell to point directly to the end of the path. This process is referred to as *path compression*. The result of path compression for type variable `ta` is illustrated in figure 13.3.3(c).

The procedure *tvar-end-value* returns the end value of a given type variable, which may be either a type construction or the representative of the type variable's congruence class; see figure 13.3.4. Since a type variable path is traversed by calling *tvar-end-value* recursively, all the type variables on the path are compressed at one time. Thus after *tvar-end-value* is called, all variables in the path headed by the given type variable refer to the end of the path. If the call to *bind-tvar!* were omitted, the type checking algorithm would be correct but less efficient, since there would be no path compression.

It is often the case that on encountering a type, one operation must be performed if it is a type construction or a type variable whose end value is a type construction, and another operation is required if it is a type variable whose end value is a congruence class representative. The procedure *type-dispatch* takes a type and two procedures, *ctvar-proc* and *tcons-proc*, and invokes the appropriate procedure with the congruence class representative or type construction, respectively, of the given type. (The `c` in `ctvar` indicates that the type variable is a congruence class representative.)

13.3.3 The Unification Algorithm

Figure 13.3.4 Type variable auxiliary procedures

```
(define tvar-end-value
  (lambda (tvar)
    (if (tvar-unbound? tvar)
      tvar
      (let ((binding (tvar-binding tvar)))
        (if (tcons? binding)
          binding
          (let ((type (tvar-end-value binding)))
            (bind-tvar! tvar type)
            type))))))

(define type-dispatch
  (lambda (type ctvar-proc tcons-proc)
    (let ((value (if (tvar? type) (tvar-end-value type) type)))
      (if (tvar? value)
        (ctvar-proc value)
        (tcons-proc value)))))
```

We are now prepared to study the unification algorithm expressed by the procedure *unify* of figure 13.3.5. If the first type passed to *unify* is a type variable whose end value is a type variable, this congruence class representative and *type2* are passed to the procedure *unify-tvar*, which we consider in a moment. If the first type is a type construction or a type variable whose end value is a type construction, we dispatch again on the second type. If it is a type variable with an end value that is an unbound type variable, *unify-tvar* is called as before. Otherwise, both types must represent type constructions (either directly or as the end values of type variables). If their type constructor names are not the same, the types cannot be unified and an error is reported. This is a common point at which the type checker detects errors. If the type constructor names are the same, the type subexpressions are unified pairwise.

The procedure *unify-tvar* takes an unbound type variable, *ctvar*, and an arbitrary type, *type*. If *type* is a type variable whose end value is also an unbound type variable, *ctvar2*, then *ctvar* is bound to *ctvar2*, or vice-versa. Since this case is symmetric, it does not matter which type variable is bound to the other. Nonetheless, since type variables that are associated with applications have less meaningful names than those associated with variables, we prefer to bind type variables associated with application so that they refer

Figure 13.3.5 A unification procedure

```
(define unify
  (lambda (type1 type2)
    (type-dispatch type1
      (lambda (ctvar1)
        (unify-tvar ctvar1 type2))
      (lambda (tcons1)
        (type-dispatch type2
          (lambda (ctvar2)
            (unify-tvar ctvar2 tcons1))
          (lambda (tcons2)
            (let ((types1 (tcons->types tcons1))
                  (types2 (tcons->types tcons2)))
              (if (and (= (length types1) (length types2))
                       (eq? (tcons->name tcons1) (tcons->name tcons2)))
                (for-each unify types1 types2)
                (type-error "Unify failure:" tcons1 tcons2)))))))))

(define unify-tvar
  (lambda (ctvar type)
    (type-dispatch type
      (lambda (ctvar2)
        (if (not (eq? ctvar ctvar2))
          (if (tvar->app? ctvar2)
            (bind-tvar! ctvar2 ctvar)
            (bind-tvar! ctvar ctvar2))))
      (lambda (tcons)
        (if (occurs-in-type? ctvar tcons)
          (type-error "Unify-tvar occur check failure:"
            ctvar tcons)
          (bind-tvar! ctvar tcons))))))

(define occurs-in-type?
  (lambda (ctvar type)
    (type-dispatch type
      (lambda (ctvar2)
        (eq? ctvar ctvar2))
      (lambda (tcons)
        (ormap (lambda (type) (occurs-in-type? ctvar type))
          (tcons->types tcons))))))
```

to type variables associated with variables, rather than the other way around. (Thus congruence class representatives will not be type variables associated with applications unless all members of the class are associated with applications.) When *tvar* and the end value of *type* are the same type variable, there is nothing to do.

If *type* is a type construction or a type variable representing a type construction, in most cases it becomes the binding of *ctvar*, but there is a possibility that must be checked first. If *type* contains as a type subexpression any type variable whose end value is *ctvar*, then there is no finite type for which *ctvar* = *type*. Consider, for example, the equation

$$t_1 = \texttt{(-> (}t_1\texttt{)}\ t_2\texttt{)}$$

There is no way to substitute a type for t_1 that will make this equation hold, since for any substitution for t_1, the right-hand side will be larger than the left-hand side. The simplest situation in which this happens is *self application*, when a procedure is applied to itself. Consider attempting to infer the type of the expression `proc (x) x(x)`. The self application `x(x)` is checked in a type environment in which `x` is associated with the unbound type variable `tx`. The check involves unifying `tx` with a newly constructed procedure type of the form `(-> (tx) t1)`.

An equation like `tx` = `(-> (tx) t1)` could be solved by allowing `tx` to be bound to an infinite type. This, however, would cause difficulties: among other things, a naive *unparse-type* would not terminate on an infinite type. Furthermore, circular types seem to arise only rarely in practice. Therefore we follow the example of most languages using type inference and regard this equation as having no solution.

To incorporate this decision in our algorithm, it is necessary for *unify-tvar* to check if *ctvar* occurs in *type* before binding *ctvar* to *type*. This is called the *occur check*, and is performed by the procedure *occurs-in-type?*. The procedure *ormap* invokes the function passed as its first argument with the elements of the list passed as its second argument until the function returns a true value, at which time *ormap* returns true, or if false is returned by the function for all elements of the list, then *ormap* returns false.

See figure 13.3.6 for the tracing output of a call to *type-of-exp* for the expression.

```
let f = proc (x) x
in f(3)
```

Figure 13.3.6 Trace of type inference algorithm

```
1 Entering with (let ((f (proc (x) x))) (f 3))
2 Entering with (proc (x) x)
3 Entering with x
2 Leaving with tx
1 Leaving with (-> (tx) tx)
  Binding tvar tf to (-> (tx) tx)
2 Entering with (f 3)
3 Entering with 3
2 Leaving with int
3 Entering with f
2 Leaving with (-> (tx) tx)
  Binding tvar t1 to tx
  Binding tvar tx to int
1 Leaving with int
0 Leaving with int
```

The entering and leaving numbers indicate the nesting of calls to *type-of-exp*. To simplify tracing output, we have turned off path compression by removing the *bind-tvar!* call in *tvar-end-value*. Expressions are unparsed to a parenthesized syntax.

• *Exercise 13.3.1*
For each of the following expressions in the language of this section, indicate its type if it is well-typed or the sort of error message that the type checker will issue if it is ill-typed. Assume + has type `(-> (int, int) int)`.

1. `(proc (x) +(x, 3))(5)`
2. `let x = 2 in let y = +(x, 5) in +(x, y)`
3. `let f = proc (x) x in if true then f(3) else f(4)`
4. `let f = proc (x) x in if f(true) then f(3) else f(4)`

□

• *Exercise 13.3.2*
Implement the type checker of this section and examine its trace output for the expressions in the previous exercise. □

• *Exercise 13.3.3*
Extend the type checker of this section to include product types, as described in exercise 13.2.1. This is challenging because one or more select statements

operating on values of some product type may be encountered before it is even known how many fields there are in the product type. A new type of the form `product*`($t_1, \ldots, t_n$) may be used to represent a record that is known to have at least n fields, but might have more than n fields. For example,

```
--> define f proc (x) +(select 1 of x, 3)
f :: (-> (product*(tfield_5, int)) int)
--> define g proc (y) if select 0 of y then select 2 of y else f(y)
g :: (-> (product*(bool, int, int)) int)
--> g( <false, 1, 2, 3> )
4 : int
--> f
<procedure> : (-> (product(bool, int, int, int)) int)
```

Here type variables created for product fields whose type is not yet known are given names of the form `tfield_`n, for some unique n. □

◦ *Exercise 13.3.4*
From the picture of binding paths in figure 13.3.3, it appears a binding path might be circular and never reach an end value. Why is this impossible? □

13.4 Polymorphic Type Inference

Some values may be interpreted as having more than one type. For example, the empty list might be the empty list of integers, or it might be the empty list of booleans. Similarly, in our typed language the procedure `proc (x) x` may be regarded as having an infinite number of types, such as `(-> (int) int)`, `(-> (bool) bool)`, and `(-> ((-> (int) bool)) (-> (int) bool))`. When such a value is used, however, it is used at only one of its types. For example, in `(cons 3 '())`, the empty list is used as a list of integers, or in our typed language, the application `(proc (x) x)(3)` uses `proc (x) x` at type `(-> (int) int)`.

Values that may be interpreted as having more than one type are said to be *polymorphic.* Both the empty list and the procedure `proc (x) x` behave in exactly the same way, regardless of the type at which they are used. In such cases polymorphism is said to be *parametric.* In other cases, the behavior of a value may depend on the type at which it is used. For example, an addition procedure might be used at the types `(-> (int, int) int)` and `(-> (real, real) real)`, among others, but employ different techniques for integer and real addition. When the behavior of a polymorphic value depends on the type at which it is used, its polymorphism is said to be *ad hoc*.

Some programming languages support *explicit* parametric polymorphism by providing means for creating and invoking procedures that take types as arguments. This approach poses difficulties, however, when combined with type inference; we do not discuss it further. An alternative approach is *implicit* parametric polymorphism, in which a type inference algorithm infers opportunities for polymorphism. Though algorithms for implicit polymorphism do not detect all possible opportunities for polymorphism, they are still quite useful and offer the advantages of type inference. In the remainder of this section we study such an algorithm.

An opportunity for parametric polymorphism is apparent when a polytype is inferred for a procedure. We have already seen the simplest example of this: the procedure `proc (x) x`. Other examples frequently arise, particularly when programming with higher-order procedures. For example, the composition procedure

```
proc (f, g)
  proc (x) f(g(x))
```

has the polytype

```
(-> ((-> (t1) t2), (-> (tx) t1))
    (-> (tx) t2))
```

Though polytypes are not types in the conventional sense associated with monotypes, they may be regarded as representing an infinite set of monotypes obtained by binding their type variables to all possible monotypes. This interpretation of type variables may be made explicit using the universal quantifier, ∀, read "for all." For example, the types of `proc (x) x` and the composition procedure may be represented, respectively, as

```
(∀tx)(-> (tx) tx)
```

and

```
(∀tx,t1,t2)(-> ((-> (t1) t2), (-> (tx) t1))        (1)
                (-> (tx) t2))
```

A polytype that has been wrapped in a universal quantifier is called a *type scheme,* since it represents an abstract structure from which types may be constructed by binding type variables. A monotype obtained by binding the type variables of a type scheme is said to be an *instance* of the type scheme.

Type schemes other than (1) might also be associated with the composition procedure, including

```
(∀t1,t2)(-> ((-> (t1) t2), (-> (int) t1))                    (2)
             (-> (int) t2))
```

and

```
(∀t1)(-> ((-> (t1) bool), (-> (int) t1))                     (3)
          (-> (int) bool))
```

Though a number of type schemes may be associated with an expression, we would like to identify one type scheme, which is "best," and regard it as the type of the expression. How do we choose the best type scheme? Observe that every instance of (3) is an instance of (2), and every instance of (2) is an instance of (1), while the converse relations do not hold. For example

```
(-> ((-> (int) int), (-> (int) int))
    (-> (int) int))                                          (4)
```

is an instance of (1) and (2), but not of (3). Evidently some type schemes are more general than others. The best type scheme to associate with an expression would be one that is most general, in the sense that every instance of any other type scheme associated with the expression is an instance of a most general type scheme. In some type disciplines there may not always be a most general type scheme, or there may be more than one of them. If there is a unique most general type, such as (1), it is said to be *principal.* A type discipline that guarantees the existence of a principal type for every expression is said to have the *principal typing property,* which is highly desirable.

In this section we will study the *Hindley-Milner type discipline*, which enjoys the principal typing property and has been popularized by the programming language ML. In this type discipline, values to which variables are bound using `let` and `letrec` are treated as polymorphic, while values that are bound during procedure application are not polymorphic. Thus, in the body of the expression

```
let comp = proc (f, g)
             proc (x) f(g(x))
in ...
```

the variable `comp` will be associated with type scheme (1). We begin our study of this algorithm by exploring the structure of type schemes.

A type variable in the body of a type scheme is said to *occur bound* if it is associated with a universal quantifier, and to *occur free* otherwise. Thus in the type `(∀t1)(-> (t1) t2)`, `t1` occurs bound and `t2` occurs free. As with α-conversion of bound variables in a procedure, the bound variables of a type scheme may be uniformly renamed without changing the meaning of the type scheme. Thus `(∀t3)(-> (t3) t2)` is the same as `(∀t1)(-> (t1) t2)`. Type variables that occur bound are said to be *generic*. It is possible for a type scheme to have none of its variables bound; in this case we say the type scheme is *unquantified*.

An instance of a type scheme is created by substituting fresh type variables for each of the generic type variables in the body of the type scheme. By convention, we assign these fresh type variables names by appending "_n" to the name of their corresponding generic type variable, where n is a unique index number. If there are multiple instances of the same generic type variable in a type scheme, they are all replaced by the same fresh type variable. Non-generic type variables in a type scheme appear in all of its instantiations without being renamed. Thus the first two instantiations of the type scheme

```
(∀t1)(-> (t1, t2) t1)
```

are `(-> (t1_1, t2) t1_1)` and `(-> (t1_2, t2) t1_2)`. The sharing relationships between the type variables of this type scheme and its instantiations are illustrated by the diagrams of figure 13.4.1.

For each use of a variable bound to a type scheme, a fresh instance of the associated type scheme is created. For example, in the program

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

the variable `id` is associated with the type scheme `(∀tx)(-> (tx) tx)`, which is instantiated to `(-> (tx_1) tx_1)` and `(-> (tx_2) tx_2)` in the first and second calls to `id`, respectively. The type variables `tx_1` and `tx_2` will be bound to `bool` and `int`, respectively, before type inference for the expression is complete.

Why do we need to distinguish generic from non-generic type variables in a type scheme? Why not simply convert a polytype into a type scheme by binding all of its variables? To see why this is unsound, consider the process of type checking the following ill-typed program.

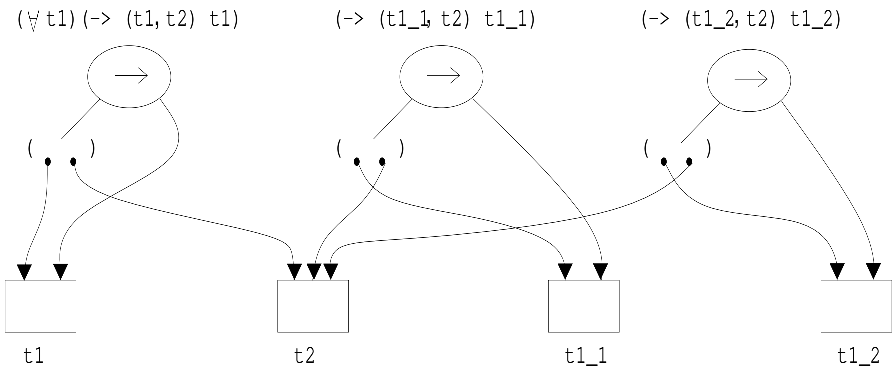

Figure 13.4.1 A type scheme and two of its instantiations.

```
let f = proc (x)
          let g = proc (y) x
          in if g(3) then g(true)
                     else +(x, 5)
in f(2)
```

First, the type environment is extended by associating `x` with an unbound type variable `tx`. The expression `proc (y) x` is then checked, in the process of which `y` is associated with another unbound type variable `ty`. Thus the polytype `(-> (ty) tx)` is obtained for the procedure `proc (y) x`. Consider the effect of naively converting this type into the type scheme

$$(\forall \texttt{tx}, \texttt{ty})\texttt{(-> (ty) tx)}$$

to obtain the type of `g` in the type environment used to check the body of the `if` expression. Separate instantiations `(-> (ty_1) tx_1)` and `(-> (ty_2) tx_2)` of the type of `g` would then be created for the applications `g(3)` and `g(true)`, respectively. In the process of checking these applications, `tx_1` and `tx_2` would then be bound to `bool` and `int`, respectively. From the application `+(x, 5)` we would then infer that `tx` must be bound to `int`. (This does not affect the reference to `tx` in the type scheme of `g`, since that reference occurs bound.) The type `(-> (int) int)` would then be inferred for `f`, so the application `f(2)` would be well-typed. Execution of this program will, however, result in a type error: the application `g(3)` returns 2, which is not a boolean value, though a test expression is required to yield a boolean value.

What went wrong in this example? The type scheme associated with `g` implies that each time `g` is called its range may be associated with any type at all, which is clearly not the case. When the type scheme of `g` was formed from the polytype `(-> (ty) tx)`, the variables `tx` and `ty` were unbound. In this case, the fact that the type variable `ty` is unbound does mean that `ty` can be associated with any type. It is not correct, however, to assume that because `tx` is unbound it may also be associated with any type. A later reference to `x` may result in `tx` being bound to a specific type, as indeed happens when the application `+(x, 5)` is checked.

This problem may be avoided by not allowing a type variable to be generic (bound by a type scheme) if it already occurs free in the current type environment. A type variable is said to *occur free in a type environment* if there are non-generic (free) occurrences of the type variable in any type that may be returned by the environment.

As a consequence of this restriction, in our last example `g` is associated with the type scheme `(∀ty)(-> (ty) tx)`. If the `if` expression's test expression is analyzed before its `then` or `else` expressions, `tx` will be bound to `bool`. An error will then be reported when either the then or else expression is analyzed, since they require that `tx` be bound to `int`. Similarly, if the test expression is checked after either the then or else expression, an error will be reported because the end value of `tx` cannot be `bool` when checking the test expression, since it has been previously bound to `int`. In either case the entire expression is rejected as ill-typed, so the runtime type error of executing an `if` expression with an integer-valued test expression is avoided.

We represent a universally quantified type as a `forall` record that includes a `body` field containing the type from which the type scheme is constructed and a `gens` field containing a list of the generic type variables occurring in the body type. See figure 13.4.2. The procedure *`build-tscheme`* constructs a type scheme from a given type, *`type`*, and environment, *`tenv`*. The generic type variables of the newly constructed type scheme are those type variables that occur in *`type`* but do not occur free in *`tenv`*. If the list of generic variables is non-empty, a new `forall` record is created; otherwise, *`type`* is returned directly.

The procedure *`instantiate-tscheme`* creates a fresh type variable for each generic type variable of a given type scheme. A copy of the type-scheme body is returned in which all references to one of its generic variables are replaced by the corresponding fresh type variable. In this copy operation it is once again critical that whenever a type variable is encountered, it is treated exactly as if its end value had appeared in its place. For each generic variable, the procedure *`instaniate-type`* is called to make a new copy of the body with

Figure 13.4.2 Type scheme procedures

```
(define-record forall (gens body))

(define build-tscheme
  (lambda (type tenv)
    (letrec ((list-gens
               (lambda (type)
                 (type-dispatch type
                   (lambda (ctvar)
                     (if (occurs-free-in-tenv? ctvar tenv)
                       '()
                       (list ctvar)))
                   (lambda (tcons)
                     (apply append
                       (map list-gens (tcons->types tcons))))))))
      (let ((gens (list-gens type)))
        (if (null? gens) type (make-forall gens type))))))

(define instantiate-tscheme
  (lambda (tscheme)
    (letrec ((loop
               (lambda (gens body)
                 (if (null? gens)
                   body
                   (loop (cdr gens)
                     (instantiate-type body (car gens)
                       (fresh-instantiated-tvar (car gens))))))))
      (loop (forall->gens tscheme) (forall->body tscheme)))))

(define instantiate-type
  (lambda (type tvar fresh-tvar)
    (letrec ((f (lambda (type)
                  (type-dispatch type
                    (lambda (ctvar)
                      (if (eq? tvar ctvar) fresh-tvar ctvar))
                    (lambda (tcons)
                      (make-tcons (tcons->name tcons)
                        (map f (tcons->types tcons))))))))
      (f type))))
```

Figure 13.4.3 Extension of the type variable ADT for polymorphism

```
(define-record tvar (cell name app? inst-count))

(define fresh-tvar
  (lambda (app? name)
    (make-tvar (make-cell '*unbound*) (string->symbol name) app? (make-cell 0))))

(define fresh-instantiated-tvar
  (lambda (tvar)
    (let ((cell (tvar->inst-count tvar)))
      (let ((count (cell-ref cell)))
        (cell-set! cell (+ 1 count))
        (fresh-tvar (tvar->app? tvar)
          (string-append (symbol->string (tvar->name tvar)) "_"
            (number->string count)))))))
```

all references to the variable replaced by its corresponding fresh variable.

The fresh type variables generated during a type scheme instantiation are provided by the procedure *fresh-instantiated-tvar*, defined in figure 13.4.3. We add to the `tvar` record an `inst-count` field, which contains a cell that in turn contains a count of the number of times that the type variable has been instantiated. This count is initialized by *fresh-tvar* and incremented by *fresh-instantiated-tvar*. The sole purpose of this count is to allow *fresh-instantiated-tvar* to generate a fresh type variable whose name is both meaningful (including the name of the type variable that it instantiates) and unique.

The type environment ADT is shown in figure 13.4.4. Each time the procedure *apply-tenv* is called, a new instantiation of the type scheme associated with a given type variable is returned. We now represent type environments as association lists. This representation makes it possible for the procedure *occurs-free-in-tenv?* to map over all the type schemes that a given environment associates with variables. In searching an environment for a free occurrence of a type variable, it is only necessary to search the unquantified schemes. This is because when we extend a type environment *tenv* with a given type scheme, the type scheme is always constructed by calling *build-tscheme* on some polytype t and that type environment *tenv*. Any type variable in t that does not already occur free in *tenv* will be made generic by *build-tscheme*. Therefore extending the type environment with the re-

Figure 13.4.4 Type environment procedures for polymorphism

```
(define extend-tenv
  (lambda (vars tschemes old-tenv)
    (append (map cons vars tschemes) old-tenv)))

(define trace-generic-variable-instantiations #t)

(define apply-tenv
  (lambda (tenv var)
    (let ((x (assq var tenv)))
      (if (pair? x)
        (let ((binding (cdr x)))
          (if (forall? binding)
            (let ((answer (instantiate-tscheme binding)))
              (if trace-generic-variable-instantiations
                (displayln
                  "  Type of var " var
                  " instantiated to " (unparse-type answer)))
              answer)
            binding))
        (error "Variable not bound:" var)))))

(define occurs-free-in-tenv?
  (lambda (ctvar tenv)
    (ormap (lambda (pair)
             (let ((binding (cdr pair)))
               (and (not (forall? binding))
                 (occurs-in-type? ctvar binding))))
      tenv)))
```

sulting type scheme will not add any new free variables to those occurring in *tenv*.

We now turn to the procedures *types-of-let-rands* and *letrec-tenv*, which provide a mechanism for the creation of polymorphic values; see figure 13.4.5. The version of *types-of-let-rands* in figure 13.4.5 differs from the version in figure 13.3.1 only in that it returns type schemes, with generic type variables quantified, rather than polytypes. Similarly, *letrec-tenv* builds a new environment in which the program variables `vars` are bound to type schemes.

Notice that the declaration expressions of a `letrec` expression are checked

Figure 13.4.5 Polymorphic type inference procedures

```
(define types-of-let-rands
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (exps (map decl->exp decls)))
      (let ((bound-variable-types (map fresh-var-tvar vars)))
        (match-types-pairwise
          bound-variable-types
          (types-of-rands exps tenv))
|       (map (lambda (type) (build-tscheme type tenv))
          bound-variable-types)))))

(define letrec-tenv
  (lambda (decls tenv)
    (let ((vars (map decl->var decls))
          (exps (map decl->exp decls)))
      (let ((bound-variable-types (map fresh-var-tvar vars)))
        (let ((new-tenv
                (extend-tenv vars bound-variable-types tenv)))
          (match-types-pairwise
            bound-variable-types
            (types-of-rands exps new-tenv))
          (extend-tenv vars
|           (map (lambda (type) (build-tscheme type tenv))
              bound-variable-types)
            tenv))))))

(define type-of-assert
  (lambda (assert-type exp tenv)
    (let ((type (build-sharing-type assert-type)))
      (match-types type (type-of-exp exp tenv))
      type)))
```

with respect to an environment in which their own types are not polymorphic. This is because it is not possible to build a type scheme for types of declaration expressions until after the expressions have been checked. Thus, inside the body of a procedure declared in a `letrec`, the types of the names declared in that `letrec` do not appear to be polymorphic. So a procedure `p` declared in a `letrec` expression, if invoked at some type t, can invoke itself only at type t, and not at, say `(list` t`)`. This restriction does not cause much difficulty in

Figure 13.4.5 Polymorphic type inference procedures (continued)

```
(define build-sharing-type
  (lambda (type)
    (let ((lookup
            (let ((var-table '()))
              (lambda (var)
                (let ((x (assq var var-table)))
                  (if (pair? x)
                    (cdr x)
                    (let ((new-tvar (fresh-var-tvar var)))
                      (set! var-table
                        (cons (cons var new-tvar) var-table))
                      new-tvar)))))))
      (letrec ((f (lambda (type)
                    (variant-case type
                      (tvarref (var) (lookup var))
                      (tcons (name types)
                        (make-tcons name (map f types)))))))
        (f type)))))
```

practice.

Finally, we consider how the procedure *type-of-assert* must be changed to accommodate polymorphism. To express the types of polymorphic values, we extend the concrete and abstract syntax of type expressions.

$\langle$type$\rangle$ ::= $\langle$tvarref$\rangle$	`tvarref (var)`

A $\langle$tvarref$\rangle$ is any $\langle$var$\rangle$ other than `int` or `bool`, or any other identifier associated with a primitive type (or type abbreviation, cf. exercise 13.1.6). We do not need concrete syntax for universal quantification of types, for we assume that any $\langle$tvarref$\rangle$ represents a generic type variable.

As we have seen, the type checker tests two type variables for equality by comparing their congruence class representatives using *eq?*. Two abstract syntax tree nodes corresponding to the same $\langle$tvarref$\rangle$ will, however, be different records containing the same symbol. This is illustrated by the following transcript, in which *t* is defined with the abstract syntax of the type `(-> (t1) t1)`.

```
> (define t
```

```
    (make-tcons '-> (list (make-tvarref 't1) (make-tvarref 't1))))
> (eq? (car (tcons->types t)) (cadr (tcons->types t)))
#f
> (eq? (tvarref->var (car (tcons->types t)))
    (tvarref->var (cadr (tcons->types t))))
#t
```

Thus it is necessary to distinguish between type variable references, as they appear in abstract syntax, and type variables as they are used by the type checker. We therefore introduce `tvarref` abstract syntax records, rather than using `tvar` records for this purpose.

The procedure *build-sharing-type* takes an abstract syntax tree representing a type and returns a corresponding type with each `tvarref` record replaced by a corresponding `tvar` record. It is necessary that all `tvarref` records containing the same ⟨var⟩ (represented by a symbol) be replaced by the same `tvar` record. To manage this, a *lookup* procedure is used that maintains a table associating symbols from the `var` fields of ⟨tvarref⟩ records with `tvar` records. The names of the newly generated type variables are obtained from the `var` field of a `tvarref` record using the same naming convention as type variables associated with lexical variable declarations.

See figure 13.4.6 for the tracing output of a call to *type-of-exp* for the following expression.

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

• *Exercise 13.4.1*

What are the types of the following expressions?

1. `let f = proc (x) x in if f(true) then f(3) else f(4)`
2. `proc (f) proc (x) f(f(x))`
3. `proc (f, x, y) f(x, y)`
4. `proc (f, x) if f(x) then f(x) else x`

□

∘ *Exercise 13.4.2*

As a consequence of restrictions on polymorphism imposed by the Hindley-Milner type discipline, some opportunities for polymorphism are lost. Give an example of a program that executes without type error, but which is rejected by the type checker of this section because a function is not polymorphic. □

Figure 13.4.6 Trace of polymorphic type inference algorithm

```
1 Entering with (let ((f (proc (x) x))) (if (f true) 3 (f 4)))
2 Entering with (proc (x) x)
3 Entering with x
2 Leaving with tx
1 Leaving with (-> (tx) tx)
  Binding tvar tf to (-> (tx) tx)
2 Entering with (if (f true) 3 (f 4))
3 Entering with (f true)
4 Entering with true
3 Leaving with bool
4 Entering with f
  Type of var f instantiated to (-> (tx_1) tx_1)
3 Leaving with (-> (tx_1) tx_1)
  Binding tvar t1 to tx_1
  Binding tvar tx_1 to bool
2 Leaving with bool
3 Entering with 3
2 Leaving with int
3 Entering with (f 4)
4 Entering with 4
3 Leaving with int
4 Entering with f
  Type of var f instantiated to (-> (tx_3) tx_3)
3 Leaving with (-> (tx_3) tx_3)
  Binding tvar t2 to tx_3
  Binding tvar tx_3 to int
2 Leaving with int
1 Leaving with int
0 Leaving with int
```

• *Exercise 13.4.3*

Implement the type checker of this section. You will need to extend the parser to recognize type variable references. Use the type checker to trace the programs in the previous exercise. □

∘ *Exercise 13.4.4*

When passed a type scheme with n generic variables, the procedure *instantiate-tscheme* of figure 13.4.2 creates n copies of the type scheme body with varying substitutions for the generic type variables. Improve the efficiency of this implementation so that only one copy of the body is made. □

∘ *Exercise 13.4.5*
Extend the type checker of this section to include product types, as described in the exercises of section 13.3.1 and 13.3.3. □

∘ *Exercise 13.4.6*
Extend the type checker of this section to include a sound sum type. This may be done by modifying `definesumtype` to generate a discriminator function in the spirit of *type-dispatch* instead of the predicate and projection functions. For example, for

```
definesumtype intlist
  emptyintlist (),
  intcons (car : int, cdr : intlist);
```

it should generate a discriminator procedure `intlistcase` of type

```
(∀t)(-> (intlist, (-> () t), (-> (int, intlist) t)) t)
```

`listcase` takes three arguments: a value of type `intlist`, a procedure of type `(-> () t)`, and a procedure of type `(-> (int, intlist) t)`. If the first argument is `emptylist`, then the first procedure is invoked. If the first argument is an `intcons`, then the second procedure is invoked on the components of the `intcons`. Note that in either case, the result is of type `t`, so type safety is preserved. □

∘ *Exercise 13.4.7*
Extend the previous exercise to allow sum types to be polymorphic. This may be done by permitting syntax such as *type-name* (*tvar* {, *tvar*}*) in place of the *type-name* in the sum type declaration. The *tvar*s may appear as *tvarref*s in the types within the declaration, and *type-name* may be used as the name of a type construction whose type operands are associated with the corresponding *tvar*s of the sum declaration. For example,

```
definesumtype list(t1)
  emptylist (),
  mycons (car : t1, cdr : (list t1))
```

□

∘ *Exercise 13.4.8*

The type inference algorithm of this section may spend considerable time in calls to `occurs-free-in-tenv`. A more efficient algorithm that avoids these calls may be constructed using the following technique. Add a *level number* field to each type variable, which is initially assigned a value that is the number of let or letrec `decl` expressions within which the subexpression being checked is nested. When a type variable with level number n is unified with a compound type t, the level number fields of all type variables in t with level numbers greater than n are assigned the value n, and if two unbound type variables are unified and their level numbers differ, the type variable with the larger level number is assigned the level number of the other type variable.

Informally justify the claim that a type variable occurs free in the current environment if and only if its level number is less than the level number associated with the declaration expression whose type scheme is being built. Extend the type checker of this section to take advantage of this technique. □

∘ *Exercise 13.4.9*

One has to be careful when assignment and polymorphic type inference are combined. Assume the primitive procedures `makecell`, `cellref`, and `cellset`, performing the cell operations used in chapter 5, are provided in the initial environment of this section's type checker and are given generic types; for example

```
makecell : (∀t)(-> (t) (cell t))
```

where `cell` is a new type constructor. Also assume `begin` is added to the language for sequencing. (Sequencing may be obtained by procedure calls, so the addition of `begin` is for convenience only.) The following program demonstrates that the type checker is unsound with these additions.

```
let cell = makecell((proc (x) x))
in begin
     cellset(cell, (proc (x) plus(x,1)));
     (cellref(cell))((proc (x) x))
   end
```

The type of `cell`, `(∀t)(cell (-> (t) t))`, is appropriate as long as the cell contains the polymorphic identity function, but not when it contains a function of type `(-> (int) int)`.

There are a variety of ways to restrict polymorphism sufficiently to stay out of this kind of trouble. The simplest is to only build a type scheme when the corresponding declaration expression is a procedure expression, a literal, or a variable reference. (Actually, as long as evaluation of a declaration expression does not result in creation of a new mutable storage location, its type may be generalized to a type scheme. It is not always possible, however, to know statically whether an expression will create mutable storage.)

Variable assignment may be simulated by binding the variable to a cell and dereferencing the cell on every reference to the variable (except in a `varassign` expression). Thus according to the above restriction, if `varassign` is added to our language, no variable that is assigned should be given a generic type.

Modify the type checker of this section to safely incorporate `begin`, `varassign`, and the cell primitive procedures. □

13.5 Summary

If a language has a static type discipline, a type checker may be used to reject ill-typed programs, so that type errors do not occur at run time. In addition, a type checker may aid compilation of efficient code. Type information generated either by the programmer or the type checker is a valuable form of documentation. The programmer must, however, understand the type discipline and may be prohibited from writing some programs that would not generate type errors if allowed to run.

Compound type expressions are built using type constructors, such as those of function types, products, and sums. A type discipline may allow new types and type abbreviations to be defined.

If the programmer is required to supply the types of all variables at the point at which they are declared, type checking can be performed by a straightforward recursive descent on the structure of the program, with a type environment recording the type of each variable in the current scope.

If types are not supplied, they may be inferred using type variables as placeholders for type information that has not yet been deduced. Types that must match are unified, which may cause type variables to be instantiated.

A polymorphic value may be interpreted as having more than one type. If a polymorphic value is always treated in the same way, its polymorphism is parametric. The Hindley-Milner type discipline infers opportunities for parametric polymorphism for values bound by `let` and `letrec`. Polymorphic types are represented by type schemes obtained from polytypes by quantifying over all type variables that do not appear free in the type environment.

Acknowledgment

This chapter includes many improvements suggested by readers of early drafts. The comments of Shriram Krishnamurthi, Gary Leavens, Greg Sullivan, and Venkatesh Choppella have been especially helpful.