Symbolic Type Widths

The intermediate code should be relatively independent of the target machine, so the optimizer does not have to change much if the code generator is replaced by one for a different machine. However, as we have described the calculation of type widths, an assumption regarding how basic types is built into the translation scheme. For instance, Example 6.12 assumes that each element of an integer array takes four bytes. Some intermediate codes, e.g., P-code for Pascal, leave it to the code generator to fill in the size of array elements, so the intermediate code is independent of the size of a machine word. We could have done the same in our translation scheme if we replaced 4 (as the width of an integer) by a symbolic constant.

b) The same as (a), but with the array stored in column-major form.

! c) An array A of k dimensions, stored in row-major form, with elements of size w. The jth dimension has indexes running from l_j to h_j.

! d) The same as (c) but with the array stored in column-major form.

Exercise 6.4.6: An integer array $A[i, j]$ has index i ranging from 1 to 10 and index j ranging from 1 to 20. Integers take 4 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[4, 5]$ b)$A[10, 8]$ c) $A[3, 17]$.

Exercise 6.4.7: Repeat Exercise 6.4.6 if A is stored in column-major order.

Exercise 6.4.8: A real array $A[i, j, k]$ has index i ranging from 1 to 4, index j ranging from 0 to 4, and index k ranging from 5 to 10. Reals take 8 bytes each. Suppose array A is stored starting at byte 0. Find the location of:

a) $A[3, 4, 5]$ b)$A[1, 2, 7]$ c) $A[4, 3, 9]$.

Exercise 6.4.9: Repeat Exercise 6.4.8 if A is stored in column-major order.

6.5 Type Checking

To do *type checking* a compiler needs to assign a type expression to each component of the source program. The compiler must then determine that these type expressions conform to a collection of logical rules that is called the *type system* for the source language.

Type checking has the potential for catching errors in programs. In principle, any check can be done dynamically, if the target code carries the type of an

element along with the value of the element. A *sound* type system eliminates the need for dynamic checking for type errors, because it allows us to determine statically that these errors cannot occur when the target program runs. An implementation of a language is *strongly typed* if a compiler guarantees that the programs it accepts will run without type errors.

Besides their use for compiling, ideas from type checking have been used to improve the security of systems that allow software modules to be imported and executed. Java programs compile into machine-independent bytecodes that include detailed type information about the operations in the bytecodes. Imported code is checked before it is allowed to execute, to guard against both inadvertent errors and malicious misbehavior.

6.5.1 Rules for Type Checking

Type checking can take on two forms: synthesis and inference. *Type synthesis* builds up the type of an expression from the types of its subexpressions. It requires names to be declared before they are used. The type of $E_1 + E_2$ is defined in terms of the types of E_1 and E_2. A typical rule for type synthesis has the form

$$\begin{array}{l}\textbf{if } f \text{ has type } s \to t \textbf{ and } x \text{ has type } s,\\ \textbf{then} \text{ expression } f(x) \text{ has type } t\end{array} \tag{6.8}$$

Here, f and x denote expressions, and $s \to t$ denotes a function from s to t. This rule for functions with one argument carries over to functions with several arguments. The rule (6.8) can be adapted for $E_1 + E_2$ by viewing it as a function application $add(E_1, E_2)$.[6]

Type inference determines the type of a language construct from the way it is used. Looking ahead to the examples in Section 6.5.4, let *null* be a function that tests whether a list is empty. Then, from the usage $null(x)$, we can tell that x must be a list. The type of the elements of x is not known; all we know is that x must be a list of elements of some type that is presently unknown.

Variables representing type expressions allow us to talk about unknown types. We shall use Greek letters $\alpha, \beta, \cdots$ for type variables in type expressions.

A typical rule for type inference has the form

$$\begin{array}{l}\textbf{if } f(x) \text{ is an expression,}\\ \textbf{then} \text{ for some } \alpha \text{ and } \beta,\ f \text{ has type } \alpha \to \beta \textbf{ and } x \text{ has type } \alpha\end{array} \tag{6.9}$$

Type inference is needed for languages like ML, which check types, but do not require names to be declared.

[6] We shall use the term "synthesis" even if some context information is used to determine types. With overloaded functions, where the same name is given to more than one function, the context of $E_1 + E_2$ may also need to be considered in some languages.

In this section, we consider type checking of expressions. The rules for checking statements are similar to those for expressions. For example, we treat the conditional statement "**if**$(E)\,S$;" as if it were the application of a function *if* to E and S. Let the special type *void* denote the absence of a value. Then function *if* expects to be applied to a *boolean* and a *void*; the result of the application is a *void*.

6.5.2 Type Conversions

Consider expressions like $x + i$, where x is of type float and i is of type integer. Since the representation of integers and floating-point numbers is different within a computer and different machine instructions are used for operations on integers and floats, the compiler may need to convert one of the operands of $+$ to ensure that both operands are of the same type when the addition occurs.

Suppose that integers are converted to floats when necessary, using a unary operator `(float)`. For example, the integer 2 is converted to a float in the code for the expression `2*3.14`:

```
t1 = (float) 2
t2 = t1 * 3.14
```

We can extend such examples to consider integer and float versions of the operators; for example, `int*` for integer operands and `float*` for floats.

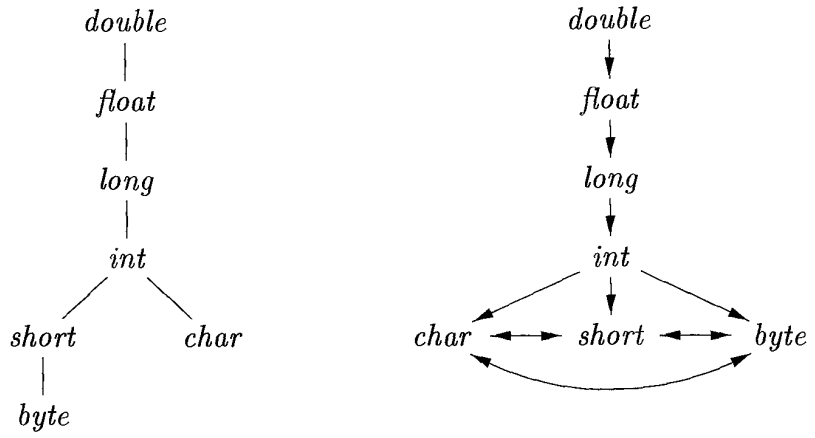
Type synthesis will be illustrated by extending the scheme in Section 6.4.2 for translating expressions. We introduce another attribute $E.type$, whose value is either *integer* or *float*. The rule associated with $E \rightarrow E_1 + E_2$ builds on the pseudocode

> **if** ($E_1.type$ = *integer* **and** $E_2.type$ = *integer*) $E.type$ = *integer*;
> **else if** ($E_1.type$ = *float* **and** $E_2.type$ = *integer*) $\cdots$
> $\cdots$

As the number of types subject to conversion increases, the number of cases increases rapidly. Therefore with large numbers of types, careful organization of the semantic actions becomes important.

Type conversion rules vary from language to language. The rules for Java in Fig. 6.25 distinguish between *widening* conversions, which are intended to preserve information, and *narrowing* conversions, which can lose information. The widening rules are given by the hierarchy in Fig. 6.25(a): any type lower in the hierarchy can be widened to a higher type. Thus, a *char* can be widened to an *int* or to a *float*, but a *char* cannot be widened to a *short*. The narrowing rules are illustrated by the graph in Fig. 6.25(b): a type s can be narrowed to a type t if there is a path from s to t. Note that *char*, *short*, and *byte* are pairwise convertible to each other.

Conversion from one type to another is said to be *implicit* if it is done automatically by the compiler. Implicit type conversions, also called *coercions*,



(a) Widening conversions (b) Narrowing conversions

Figure 6.25: Conversions between primitive types in Java

are limited in many languages to widening conversions. Conversion is said to be *explicit* if the programmer must write something to cause the conversion. Explicit conversions are also called *casts*.

The semantic action for checking $E \rightarrow E_1 + E_2$ uses two functions:

1. $max(t_1, t_2)$ takes two types t_1 and t_2 and returns the maximum (or least upper bound) of the two types in the widening hierarchy. It declares an error if either t_1 or t_2 is not in the hierarchy; e.g., if either type is an array or a pointer type.

2. $widen(a, t, w)$ generates type conversions if needed to widen an address a of type t into a value of type w. It returns a itself if t and w are the same type. Otherwise, it generates an instruction to do the conversion and place the result in a temporary t, which is returned as the result. Pseudocode for *widen*, assuming that the only types are *integer* and *float*, appears in Fig. 6.26.

```
Addr widen(Addr a, Type t, Type w)
        if ( t = w ) return a;
        else if ( t = integer and w = float ) {
                temp = new Temp();
                gen(temp '=' '(float)' a);
                return temp;
        }
        else error;
}
```

Figure 6.26: Pseudocode for function *widen*

The semantic action for $E \rightarrow E_1 + E_2$ in Fig. 6.27 illustrates how type conversions can be added to the scheme in Fig. 6.20 for translating expressions. In the semantic action, temporary variable a_1 is either $E_1.addr$, if the type of E_1 does not need to be converted to the type of E, or a new temporary variable returned by *widen* if this conversion is necessary. Similarly, a_2 is either $E_2.addr$ or a new temporary holding the type-converted value of E_2. Neither conversion is needed if both types are *integer* or both are *float*. In general, however, we could find that the only way to add values of two different types is to convert them both to a third type.

$$
\begin{array}{lll}
E \rightarrow E_1 + E_2 & \{ & E.type = max(E_1.type, E_2.type); \\
 & & a_1 = widen(E_1.addr, E_1.type, E.type); \\
 & & a_2 = widen(E_2.addr, E_2.type, E.type); \\
 & & E.addr = \textbf{new } Temp\,(); \\
 & & gen(E.addr\ '='\ a_1\ '+'\ a_2); \ \}
\end{array}
$$

Figure 6.27: Introducing type conversions into expression evaluation

6.5.3 Overloading of Functions and Operators

An *overloaded* symbol has different meanings depending on its context. Overloading is *resolved* when a unique meaning is determined for each occurrence of a name. In this section, we restrict attention to overloading that can be resolved by looking only at the arguments of a function, as in Java.

Example 6.13: The + operator in Java denotes either string concatenation or addition, depending on the types of its operands. User-defined functions can be overloaded as well, as in

```
void err() { ··· }
void err(String s) { ··· }
```

Note that we can choose between these two versions of a function `err` by looking at their arguments. □

The following is a type-synthesis rule for overloaded functions:

if f can have type $s_i \rightarrow t_i$, for $1 \leq i \leq n$, where $s_i \neq s_j$ for $i \neq j$
and x has type s_k, for some $1 \leq k \leq n$ (6.10)
then expression $f(x)$ has type t_k

The value-number method of Section 6.1.2 can be applied to type expressions to resolve overloading based on argument types, efficiently. In a DAG representing a type expression, we assign an integer index, called a value number, to each node. Using Algorithm 6.3, we construct a signature for a node,

consisting of its label and the value numbers of its children, in order from left to right. The signature for a function consists of the function name and the types of its arguments. The assumption that we can resolve overloading based on the types of arguments is equivalent to saying that we can resolve overloading based on signatures.

It is not always possible to resolve overloading by looking only at the arguments of a function. In Ada, instead of a single type, a subexpression standing alone may have a set of possible types for which the context must provide sufficient information to narrow the choice down to a single type (see Exercise 6.5.2).

6.5.4 Type Inference and Polymorphic Functions

Type inference is useful for a language like ML, which is strongly typed, but does not require names to be declared before they are used. Type inference ensures that names are used consistently.

The term "polymorphic" refers to any code fragment that can be executed with arguments of different types. In this section, we consider *parametric polymorphism*, where the polymorphism is characterized by parameters or type variables. The running example is the ML program in Fig. 6.28, which defines a function *length*. The type of *length* can be described as, "for any type α, *length* maps a list of elements of type α to an integer."

fun *length*(x) =
    if *null*(x) **then** 0 **else** *length*(*tl*(x)) + 1;

Figure 6.28: ML program for the length of a list

Example 6.14: In Fig. 6.28, the keyword **fun** introduces a function definition; functions can be recursive. The program fragment defines function *length* with one parameter x. The body of the function consists of a conditional expression. The predefined function *null* tests whether a list is empty, and the predefined function *tl* (short for "tail") returns the remainder of a list after the first element is removed.

The function *length* determines the length or number of elements of a list x. All elements of a list must have the same type, but *length* can be applied to lists whose elements are of any one type. In the following expression, *length* is applied to two different types of lists (list elements are enclosed within "[" and "]"):

$$length([\texttt{"sun"}, \texttt{"mon"}, \texttt{"tue"}]) + length([10, 9, 8, 7]) \tag{6.11}$$

The list of strings has length 3 and the list of integers has length 4, so expression (6.11) evaluates to 7. □

Using the symbol $\forall$ (read as "for any type") and the type constructor *list*, the type of *length* can be written as

$$\forall\alpha.\ list(\alpha) \rightarrow integer \tag{6.12}$$

The $\forall$ symbol is the *universal quantifier*, and the type variable to which it is applied is said to be *bound* by it. Bound variables can be renamed at will, provided all occurrences of the variable are renamed. Thus, the type expression

$$\forall\beta.\ list(\beta) \rightarrow integer$$

is equivalent to (6.12). A type expression with a $\forall$ symbol in it will be referred to informally as a "polymorphic type."

Each time a polymorphic function is applied, its bound type variables can denote a different type. During type checking, at each use of a polymorphic type we replace the bound variables by fresh variables and remove the universal quantifiers.

The next example informally infers a type for *length*, implicitly using type inference rules like (6.9), which is repeated here:

> **if** $f(x)$ is an expression,
> **then** for some α and β, f has type $\alpha \rightarrow \beta$ **and** x has type α

Example 6.15: The abstract syntax tree in Fig. 6.29 represents the definition of *length* in Fig. 6.28. The root of the tree, labeled **fun**, represents the function definition. The remaining nonleaf nodes can be viewed as function applications. The node labeled + represents the application of the operator + to a pair of children. Similarly, the node labeled **if** represents the application of an operator **if** to a triple formed by its children (for type checking, it does not matter that either the **then** or the **else** part will be evaluated, but not both).

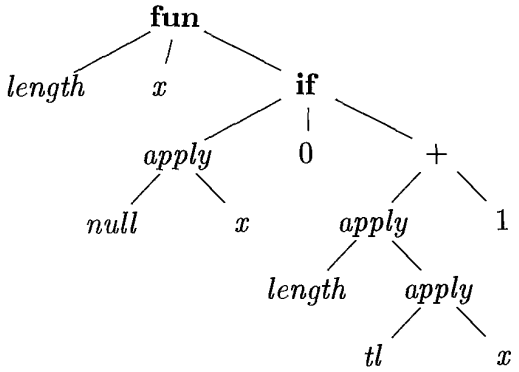


Figure 6.29: Abstract syntax tree for the function definition in Fig. 6.28

From the body of function *length*, we can infer its type. Consider the children of the node labeled **if**, from left to right. Since *null* expects to be applied to lists, x must be a list. Let us use variable α as a placeholder for the type of the list elements; that is, x has type "list of α."

Substitutions, Instances, and Unification

If t is a type expression and S is a substitution (a mapping from type variables to type expressions), then we write $S(t)$ for the result of consistently replacing all occurrences of each type variable α in t by $S(\alpha)$. $S(t)$ is called an *instance* of t. For example, *list(integer)* is an instance of *list*(α), since it is the result of substituting *integer* for α in *list*(α). Note, however, that *integer* $\rightarrow$ *float* is not an instance of $\alpha \rightarrow \alpha$, since a substitution must replace all occurrences of α by the same type expression.

Substitution S is a *unifier* of type expressions t_1 and t_2 if $S(t_1) = S(t_2)$. S is the *most general unifier* of t_1 and t_2 if for any other unifier of t_1 and t_2, say S', it is the case that for any t, $S'(t)$ is an instance of $S(t)$. In words, S' imposes more constraints on t than S does.

If *null*(x) is true, then *length*(x) is 0. Thus, the type of *length* must be "function from list of α to integer." This inferred type is consistent with the usage of *length* in the else part, *length*(*tl*(x)) + 1. □

Since variables can appear in type expressions, we have to re-examine the notion of equivalence of types. Suppose E_1 of type $s \rightarrow s'$ is applied to E_2 of type t. Instead of simply determining the equality of s and t, we must "unify" them. Informally, we determine whether s and t can be made structurally equivalent by replacing the type variables in s and t by type expressions.

A *substitution* is a mapping from type variables to type expressions. We write $S(t)$ for the result of applying the substitution S to the variables in type expression t; see the box on "Substitutions, Instances, and Unification." Two type expressions t_1 and t_2 *unify* if there exists some substitution S such that $S(t_1) = S(t_2)$. In practice, we are interested in the most general unifier, which is a substitution that imposes the fewest constraints on the variables in the expressions. See Section 6.5.5 for a unification algorithm.

Algorithm 6.16: Type inference for polymorphic functions.

INPUT: A program consisting of a sequence of function definitions followed by an expression to be evaluated. An expression is made up of function applications and names, where names can have predefined polymorphic types.

OUTPUT: Inferred types for the names in the program.

METHOD: For simplicity, we shall deal with unary functions only. The type of a function $f(x_1, x_2)$ with two parameters can be represented by a type expression $s_1 \times s_2 \rightarrow t$, where s_1 and s_2 are the types of x_1 and x_2, respectively, and t is the type of the result $f(x_1, x_2)$. An expression $f(a, b)$ can be checked by matching the type of a with s_1 and the type of b with s_2.

Check the function definitions and the expression in the input sequence. Use the inferred type of a function if it is subsequently used in an expression.

- For a function definition **fun** $\mathbf{id}_1(\mathbf{id}_2) = E$, create fresh type variables α and β. Associate the type $\alpha \rightarrow \beta$ with the function $\mathbf{id}_1$, and the type α with the parameter $\mathbf{id}_2$. Then, infer a type for expression E. Suppose α denotes type s and β denotes type t after type inference for E. The inferred type of function $\mathbf{id}_1$ is $s \rightarrow t$. Bind any type variables that remain unconstrained in $s \rightarrow t$ by $\forall$ quantifiers.

- For a function application $E_1(E_2)$, infer types for E_1 and E_2. Since E_1 is used as a function, its type must have the form $s \rightarrow s'$. (Technically, the type of E_1 must unify with $\beta \rightarrow \gamma$, where β and γ are new type variables). Let t be the inferred type of E_1. Unify s and t. If unification fails, the expression has a type error. Otherwise, the inferred type of $E_1(E_2)$ is s'.

- For each occurrence of a polymorphic function, replace the bound variables in its type by distinct fresh variables and remove the $\forall$ quantifiers. The resulting type expression is the inferred type of this occurrence.

- For a name that is encountered for the first time, introduce a fresh variable for its type.

□

Example 6.17: In Fig. 6.30, we infer a type for function *length*. The root of the syntax tree in Fig. 6.29 is for a function definition, so we introduce variables β and γ, associate the type $\beta \rightarrow \gamma$ with function *length*, and the type β with x; see lines 1-2 of Fig. 6.30.

At the right child of the root, we view **if** as a polymorphic function that is applied to a triple, consisting of a boolean and two expressions that represent the **then** and **else** parts. Its type is $\forall\alpha.\ \mathit{boolean} \times \alpha \times \alpha \rightarrow \alpha$.

Each application of a polymorphic function can be to a different type, so we make up a fresh variable α_i (where i is from "if") and remove the $\forall$; see line 3 of Fig. 6.30. The type of the left child of **if** must unify with *boolean*, and the types of its other two children must unify with α_i.

The predefined function *null* has type $\forall\alpha.\ \mathit{list}(\alpha) \rightarrow \mathit{boolean}$. We use a fresh type variable α_n (where n is for "null") in place of the bound variable α; see line 4. From the application of *null* to x, we infer that the type β of x must match $\mathit{list}(\alpha_n)$; see line 5.

At the first child of **if**, the type *boolean* for $\mathit{null}(x)$ matches the type expected by **if**. At the second child, the type α_i unifies with *integer*; see line 6.

Now, consider the subexpression $\mathit{length}(\mathit{tl}(x)) + 1$. We make up a fresh variable α_t (where t is for "tail") for the bound variable α in the type of *tl*; see line 8. From the application $\mathit{tl}(x)$, we infer $\mathit{list}(\alpha_t) = \beta = \mathit{list}(\alpha_n)$; see line 9.

Since $\mathit{length}(\mathit{tl}(x))$ is an operand of $+$, its type γ must unify with *integer*; see line 10. It follows that the type of *length* is $\mathit{list}(\alpha_n) \rightarrow \mathit{integer}$. After the

LINE	EXPRESSION : TYPE	UNIFY
1)	$length : \beta \rightarrow \gamma$	
2)	$x : \beta$	
3)	$\textbf{if} : boolean \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
4)	$null : list(\alpha_n) \rightarrow boolean$	
5)	$null(x) : boolean$	$list(\alpha_n) = \beta$
6)	$0 : integer$	$\alpha_i = integer$
7)	$+ : integer \times integer \rightarrow integer$	
8)	$tl : list(\alpha_t) \rightarrow list(\alpha_t)$	
9)	$tl(x) : list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
10)	$length(tl(x)) : \gamma$	$\gamma = integer$
11)	$1 : integer$	
12)	$length(tl(x)) + 1 : integer$	
13)	$\textbf{if}(\cdots) : integer$	

Figure 6.30: Inferring a type for the function *length* of Fig. 6.28

function definition is checked, the type variable α_n remains in the type of *length*. Since no assumptions were made about α_n, any type can be substituted for it when the function is used. We therefore make it a bound variable and write

$$\forall \alpha_n.\ list(\alpha_n) \rightarrow integer$$

for the type of *length*. □

6.5.5 An Algorithm for Unification

Informally, unification is the problem of determining whether two expressions s and t can be made identical by substituting expressions for the variables in s and t. Testing equality of expressions is a special case of unification; if s and t have constants but no variables, then s and t unify if and only if they are identical. The unification algorithm in this section extends to graphs with cycles, so it can be used to test structural equivalence of circular types.[7]

We shall implement a graph-theoretic formulation of unification, where types are represented by graphs. Type variables are represented by leaves and type constructors are represented by interior nodes. Nodes are grouped into equivalence classes; if two nodes are in the same equivalence class, then the type expressions they represent must unify. Thus, all interior nodes in the same class must be for the same type constructor, and their corresponding children must be equivalent.

Example 6.18: Consider the two type expressions

[7] In some applications, it is an error to unify a variable with an expression containing that variable. Algorithm 6.19 permits such substitutions.

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_3)) \rightarrow list(\alpha_2)$$
$$((\alpha_3 \rightarrow \alpha_4) \times list(\alpha_3)) \rightarrow \alpha_5$$

The following substitution S is the most general unifier for these expressions

x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$list(\alpha_2)$

This substitution maps the two type expressions to the following expression

$$((\alpha_1 \rightarrow \alpha_2) \times list(\alpha_1)) \rightarrow list(\alpha_2)$$

The two expressions are represented by the two nodes labeled $\rightarrow$: 1 in Fig. 6.31. The integers at the nodes indicate the equivalence classes that the nodes belong to after the nodes numbered 1 are unified. □

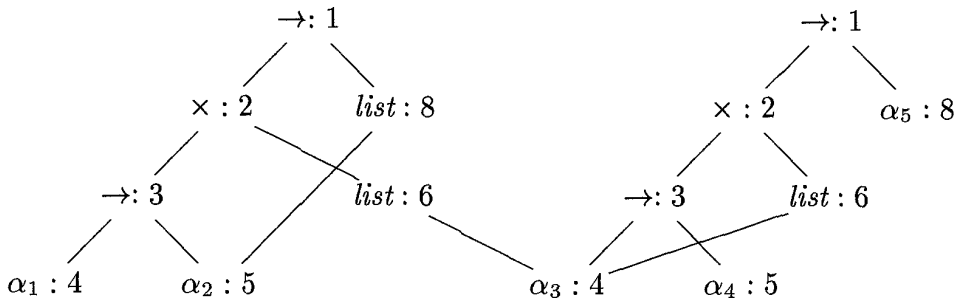


Figure 6.31: Equivalence classes after unification

Algorithm 6.19: Unification of a pair of nodes in a type graph.

INPUT: A graph representing a type and a pair of nodes m and n to be unified.

OUTPUT: Boolean value true if the expressions represented by the nodes m and n unify; false, otherwise.

METHOD: A node is implemented by a record with fields for a binary operator and pointers to the left and right children. The sets of equivalent nodes are maintained using the *set* field. One node in each equivalence class is chosen to be the unique representative of the equivalence class by making its *set* field contain a null pointer. The *set* fields of the remaining nodes in the equivalence class will point (possibly indirectly through other nodes in the set) to the representative. Initially, each node n is in an equivalence class by itself, with n as its own representative node.

The unification algorithm, shown in Fig. 6.32, uses the following two operations on nodes:

```
boolean unify(Node m, Node n) {
        s = find(m); t = find(n);
        if ( s = t ) return true;
        else if ( nodes s and t represent the same basic type ) return true;
        else if (s is an op-node with children s1 and s2 and
                        t is an op-node with children t1 and t2) {
                union(s, t);
                return unify(s1, t1) and unify(s2, t2);
        }
        else if s or t represents a variable {
                union(s, t);
                return true;
        }
        else return false;
}
```

Figure 6.32: Unification algorithm.

- *find*(n) returns the representative node of the equivalence class currently containing node n.

- *union*(m, n) merges the equivalence classes containing nodes m and n. If one of the representatives for the equivalence classes of m and n is a nonvariable node, *union* makes that nonvariable node be the representative for the merged equivalence class; otherwise, *union* makes one or the other of the original representatives be the new representative. This asymmetry in the specification of *union* is important because a variable cannot be used as the representative for an equivalence class for an expression containing a type constructor or basic type. Otherwise, two inequivalent expressions may be unified through that variable.

The *union* operation on sets is implemented by simply changing the *set* field of the representative of one equivalence class so that it points to the representative of the other. To find the equivalence class that a node belongs to, we follow the *set* pointers of nodes until the representative (the node with a null pointer in the set field) is reached.

Note that the algorithm in Fig. 6.32 uses $s =$ *find*(m) and $t =$ *find*(n) rather than m and n, respectively. The representative nodes s and t are equal if m and n are in the same equivalence class. If s and t represent the same basic type, the call *unify*(m, n) returns true. If s and t are both interior nodes for a binary type constructor, we merge their equivalence classes on speculation and recursively check that their respective children are equivalent. By merging first, we decrease the number of equivalence classes before recursively checking the children, so the algorithm terminates.

The substitution of an expression for a variable is implemented by adding the leaf for the variable to the equivalence class containing the node for that expression. Suppose either m or n is a leaf for a variable. Suppose also that this leaf has been put into an equivalence class with a node representing an expression with a type constructor or a basic type. Then *find* will return a representative that reflects that type constructor or basic type, so that a variable cannot be unified with two different expressions. □

Example 6.20: Suppose that the two expressions in Example 6.18 are represented by the initial graph in Fig. 6.33, where each node is in its own equivalence class. When Algorithm 6.19 is applied to compute *unify*$(1, 9)$, it notes that nodes 1 and 9 both represent the same operator. It therefore merges 1 and 9 into the same equivalence class and calls *unify*$(2, 10)$ and *unify*$(8, 14)$. The result of computing *unify*$(1, 9)$ is the graph previously shown in Fig. 6.31. □

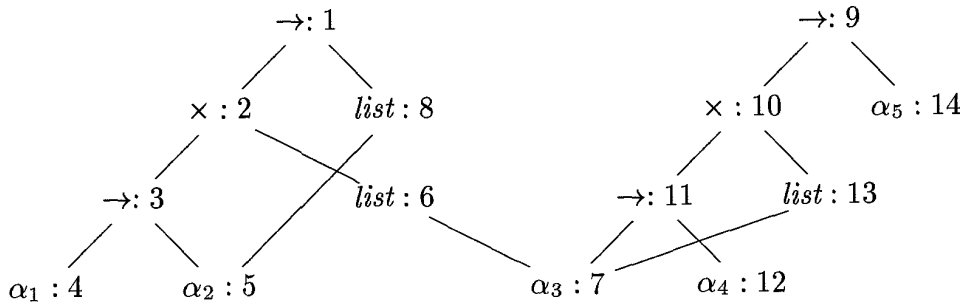


Figure 6.33: Initial graph with each node in its own equivalence class

If Algorithm 6.19 returns true, we can construct a substitution S that acts as the unifier, as follows. For each variable α, *find*(α) gives the node n that is the representative of the equivalence class of α. The expression represented by n is $S(\alpha)$. For example, in Fig. 6.31, we see that the representative for α_3 is node 4, which represents α_1. The representative for α_5 is node 8, which represents $list(\alpha_2)$. The resulting substitution S is as in Example 6.18.

6.5.6 Exercises for Section 6.5

Exercise 6.5.1: Assuming that function *widen* in Fig. 6.26 can handle any of the types in the hierarchy of Fig. 6.25(a), translate the expressions below. Assume that c and d are characters, s and t are short integers, i and j are integers, and x is a float.

a) `x = s + c.`

b) `i = s + c.`

c) `x = (s + c) * (t + d).`

Exercise 6.5.2: As in Ada, suppose that each expression must have a unique type, but that from a subexpression, by itself, all we can deduce is a set of possible types. That is, the application of function E_1 to argument E_2, represented by $E \rightarrow E_1 (E_2)$, has the associated rule

$$E.type = \{ t \mid \text{for some } s \text{ in } E_2.type,\ s \rightarrow t \text{ is in } E_1.type \}$$

Describe an SDD that determines a unique type for each subexpression by using an attribute *type* to synthesize a set of possible types bottom-up, and, once the unique type of the overall expression is determined, proceeds top-down to determine attribute *unique* for the type of each subexpression.

6.6 Control Flow

The translation of statements such as if-else-statements and while-statements is tied to the translation of boolean expressions. In programming languages, boolean expressions are often used to

1. *Alter the flow of control.* Boolean expressions are used as conditional expressions in statements that alter the flow of control. The value of such boolean expressions is implicit in a position reached in a program. For example, in **if** (E) S, the expression E must be true if statement S is reached.

2. *Compute logical values.* A boolean expression can represent *true* or *false* as values. Such boolean expressions can be evaluated in analogy to arithmetic expressions using three-address instructions with logical operators.

The intended use of boolean expressions is determined by its syntactic context. For example, an expression following the keyword **if** is used to alter the flow of control, while an expression on the right side of an assignment is used to denote a logical value. Such syntactic contexts can be specified in a number of ways: we may use two different nonterminals, use inherited attributes, or set a flag during parsing. Alternatively we may build a syntax tree and invoke different procedures for the two different uses of boolean expressions.

This section concentrates on the use of boolean expressions to alter the flow of control. For clarity, we introduce a new nonterminal B for this purpose. In Section 6.6.6, we consider how a compiler can allow boolean expressions to represent logical values.

6.6.1 Boolean Expressions

Boolean expressions are composed of the boolean operators (which we denote &&, | |, and !, using the C convention for the operators AND, OR, and NOT, respectively) applied to elements that are boolean variables or relational expressions. Relational expressions are of the form E_1 **rel** E_2, where E_1 and