8.5 Type Equivalence

A major question involved in the application of types to type checking has to do with type equivalence: When are two types the same? One way of trying to answer this question is to compare the sets of values simply as sets. Two sets are the same if they contain the same values: For example, any type defined as the Cartesian product $A \times B$ is the same as any other type defined in the same way. On the other hand, if we assume that type $B$ is not the same as type $A$, then a type defined as $A \times B$ is not the same as a type defined as $B \times A$, since $A \times B$ contains the pairs $(a, b)$ but $B \times A$ contains the pairs $(b, a)$. This view of type equivalence is that two data types are the same if they have the same structure: They are built in exactly the same way using the same type constructors from the same simple types. This form of type equivalence is called structural equivalence and is one of the principal forms of type equivalence in programming languages.

Example

In a C-like syntax, the types `struct Rec1` and `struct Rec2` defined as follows are structurally equivalent, but `struct Rec1` and `struct Rec3` are not (the `char` and `int` fields are reversed in the definition of `struct Rec3`):

```c
struct Rec1{
    char x;
    int y;
    char z[10];
};
```

(continues)
Structural equivalence is relatively easy to implement (except for recursive types, as explained later in this chapter). What’s more, it provides all the information needed to perform error checking and storage allocation. It is used in such languages as Algol60, Algol68, FORTRAN, COBOL, and a few modern languages such as Modula-3. It is also used selectively in such languages as C, Java, and ML, as explained later in this chapter. To check structural equivalence, a translator may represent types as trees and check equivalence recursively on subtrees (see Exercise 8.36). Questions still arise, however, in determining how much information is included in a type under the application of a type constructor. For example, are the two types $A_1$ and $A_2$ defined by

```c
typedef int A1[10];
typedef int A2[20];
```

structurally equivalent? Perhaps yes, if the size of the index set is not part of an array type; otherwise, no. A similar question arises with respect to member names of structures. If structures are taken to be just Cartesian products, then, for example, the two structures

```c
struct RecA{
  char x;
  int y;
};
```

and

```c
struct RecB{
  char a;
  int b;
};
```

should be structurally equivalent; typically, however, they are not equivalent, because variables of the different structures would have to use different names to access the member data.
A complicating factor is the use of type names in declarations. As noted previously, type expressions in declarations may or may not be given explicit names. For example, in C, variable declarations may be given using anonymous types (type constructors applied without giving them names), but names can also be given right in structs and unions, or by using a typedef. For example, consider the C code:

```c
struct RecA{
    char x;
    int y;
} a;

typedef struct RecA RecA;

typedef struct{
    char x;
    int y;
} RecB;

RecB b;

struct{
    char x;
    int y;
} c;
```

Variable `a` has a data type with two names: struct RecA and RecA (as given by the typedef). Variable `b`'s type has only the name RecB (the struct name was left blank). And variable `c`'s type has no name at all! (Actually, `c`'s type still has a name, but it is internal and cannot be referred to by the programmer). Of course, the types struct RecA, RecA, RecB, and `c`'s anonymous type are all structurally equivalent.

Structural equivalence in the presence of type names remains relatively simple—simply replace each name by its associated type expression in its declaration—except for recursive types, where this rule would lead to an infinite loop (recall that a major reason for introducing type names is to allow for the declaration of recursive types). Consider the following example (a variation on an example used previously):

```c
typedef struct CharListNode* CharList;
typedef struct CharListNode2* CharList2;

struct CharListNode{
    char data;
    CharList next;
};

struct CharListNode2{
    char data;
    CharList2 next;
};
```
Clearly, CharList and CharList2 are structurally equivalent, but a type checker that simply replaces type names by definitions will get into an infinite loop trying to verify it! The secret is to assume that CharList and CharList2 are structurally equivalent to start with. It then follows easily that CharListNode and CharListNode2 are structurally equivalent, and then that CharList and CharList2 are indeed themselves equivalent. Of course, this seems like a circular argument, and it must be done carefully to give correct results. The details can be found in Louden [1997].

To avoid this problem, a different, much stricter, type equivalence algorithm was developed that focuses on the type names themselves. Two types are the same only if they have the same name. For obvious reasons, this is called name equivalence.

Example
In the following C declarations,

```c
struct RecA{
    char x;
    int y;
};

typedef struct RecA RecA;

struct RecA a;
RecA b;
struct RecA c;
struct{
    char x;
    int y;
} d;
```

all of the variables a, b, c, and d are structurally equivalent. However, a and c are name equivalent (and not name equivalent to b or d), while b and d are not name equivalent to any other variable. Similarly, given the declarations

```c
typedef int Ar1[10];
typedef Ar1 Ar2;
typedef int Age;
```

types Ar1 and Ar2 are structurally equivalent but not name equivalent, and Age and int are structurally equivalent but not name equivalent.

Name equivalence in its purest form is even easier to implement than structural equivalence, as long as we force every type to have an explicit name (this can actually be a good idea, since it documents every type with an explicit name): Two types are the same only if they are the same name, and two variables are type equivalent only if their declarations use exactly the same type name. We can also invent aliases for types (e.g., Age above is an alias for int), and the type checker forces the programmer to keep uses of aliases distinct (this is also a very good design tool).
The situation becomes slightly more complex if we allow variable or function declarations to contain new types (i.e., type constructors) rather than existing type names only. Consider, for example, the following declarations:

```c
struct{
    char x;
    int y;
} d,e;
```

Are `d` and `e` name equivalent? Here there are no visible type names from which to form a conclusion, so a name equivalence algorithm could say either yes or no. A language might include a rule that a combined declaration as this is equivalent to separate declarations:

```c
struct{
    char x;
    int y;
} d;
struct{
    char x;
    int y;
} e;
```

In this case, new internal names are generated for each new `struct`, so `d` and `e` are clearly not name equivalent. On the other hand, using a single `struct` in a combined declaration could be viewed as constructing only one internal name, in which case `d` and `e` are equivalent.

Ada is one of the few languages to implement a very pure form of name equivalence. The only time Ada allows type constructors to be used in variable declarations is with the array constructor:

```ada
a: array (1..10) of integer;
```

The following is illegal Ada:

```ada
a: record
    x: integer;
    y: character;
end record;
```

Instead, we must write:

```ada
type IntChar is record
    x: integer;
    y: character;
end record;
a: IntChar;
```
Thus, ambiguity is avoided in Ada by requiring type names in variable and function declarations in virtually all cases. Ada views simultaneous array variable declarations without type names as having separate, inequivalent types.

Note that one small special rule is made in Ada for aliases and subtypes. According to this rule, writing:

```ada
type Age is integer;
```

is illegal. The Ada compiler wants to know whether we want an actual new type or just an alias that should be considered equivalent to the old type. Here we must write either:

```ada
type Age is new integer;
   -- Age is a completely new type
```
or:

```ada
subtype Age is integer;
   -- Age is just an alias for integer
```

As noted previously, the `subtype` designation is also used to create subset types, which are also not new types, but indications for runtime value checking.

C uses a form of type equivalence that falls between name and structural equivalence, and which can be loosely described as “name equivalence for structs and unions, structural equivalence for everything else.” What is really meant here is that applying the `struct` or `union` type constructor creates a new, nonequivalent, type. Applying any other type constructor, or using a `typedef`, does not create a new type but one that is equivalent to every other type with the same structure (taking into account the special rule for structs and unions).

**Example**

```ada
struct A{
   char x;
   int y;
};

struct B{
   char x;
   int y;
};

typedef struct A C;
typedef C* P;
typedef struct B * Q;
typedef struct A * R;
typedef int S[10];
```

(continues)
(continued)

```c
typedef int T[5];
typedef int Age;
typedef int (*F)(int);
typedef Age (*G)(Age);
```

Types `struct A` and `struct B` are equivalent, but they are not equivalent to `struct E`. Types `P` and `R` are equivalent, but not to `Q`; types `S` and `T` are equivalent; types `int` and `Age` are equivalent, as are function types `P` and `G`.

Pascal adopts a similar rule to C, except that almost all type constructors, including arrays, pointers, and enumerations, lead to new, inequivalent types. However, new names for existing type names are, as in C’s `typedef`, equivalent to the original types. (Sometimes this rule is referred to as declaration equivalence.) Thus, in:

```pascal

type
  IntPtr = ^integer;
  Age = integer;
var
  x: IntPtr;
  y: ^integer;
  i: Age;
  a: integer;
```

`x` and `y` are not equivalent in Pascal, but `i` and `a` are.

Java has a particularly simple approach to type equivalence. First, there are no `typedefs`, so naming questions are minimized. Second, `class` and `interface` declarations implicitly create new type names (the same as the class/interface names), and name equivalence is used for these types. The only complication is that arrays (which cannot have type names) use structural equivalence, with special rules for establishing base type equivalence (we do not further discuss Java arrays in this text—see the Notes and References for more information).

Last, we mention the type equivalence rule used in ML. ML has two type declarations—`datatype` and `type`, but only the former constructs a new type, while the latter only constructs aliases of existing types (like the `typedef` of C). For example,

```ml

type Age = int;
datatype NewAge = NewAge int;
```

Now `Age` is equivalent to `int`, but `NewAge` is a new type not equivalent to `int`. Indeed, we reused the name `NewAge` to also stand for a data constructor for the `NewAge` type, so that `(NewAge 2)` is a value of type `NewAge`, while `2` is a value of type `int` (or `Age`).

One issue we have omitted from this discussion of type equivalence is the interaction between the type equivalence algorithm and the type-checking algorithm. As explained earlier, type equivalence

---

\[\text{Subrange types in Pascal are implemented as runtime checks, not new types.}\]
comes up in type checking and can be given a somewhat independent answer. In fact, some type equivalence questions may never arise because of the type-checking rules, and so we might never be able to write code in a language that would answer a particular type equivalence question. In such cases, we might sensibly adopt an operational view and simply disregard the question of type equivalence for these particular cases. An example of this is discussed in Exercise 8.25.

### 8.6 Type Checking

Type checking, as explained at the beginning of the chapter, is the process by which a translator verifies that all constructs in a program make sense in terms of the types of its constants, variables, procedures, and other entities. It involves the application of a type equivalence algorithm to expressions and statements, with the type-checking algorithm varying the use of the type equivalence algorithm to suit the context. Thus, a strict type equivalence algorithm such as name equivalence could be relaxed by the type checker if the situation warrants.

Type checking can be divided into dynamic and static checking. If type information is maintained and checked at runtime, the checking is dynamic. Interpreters by definition perform dynamic type checking. But compilers can also generate code that maintains type attributes during runtime in a table or as type tags in an environment. A Lisp compiler, for example, would do this. Dynamic type checking is required when the types of objects can only be determined at runtime.

The alternative to dynamic typing is static typing: The types of expressions and objects are determined from the text of the program, and type checking is performed by the translator before execution. In a strongly typed language, all type errors must be caught before runtime, so these languages must be statically typed, and type errors are reported as compilation error messages that prevent execution. However, a language definition may leave unspecified whether dynamic or static typing is to be used.

**Example 1**

C compilers apply static type checking during translation, but C is not really strongly typed since many type inconsistencies do not cause compilation errors but are automatically removed through the generation of conversion code, either with or without a warning message. Most modern compilers, however, have error level settings that do provide stronger typing if it is desired. C++ also adds stronger type checking to C, but also mainly in the form of compiler warnings rather than errors (for compatibility with C). Thus, in C++ (and to a certain extent also in C), many type errors appear only as warnings and do not prevent execution. Thus, ignoring warnings can be a “dangerous folly” (Stroustrup [1994], p. 42).

**Example 2**

The Scheme dialect of Lisp (see Chapter 3) is a dynamically typed language, but types are rigorously checked, with all type errors causing program termination. There are no types in declarations, and there are no explicit type names. Variables and other symbols have no predeclared types but take on the type of the value they possess at each moment of execution. Thus, types in Scheme must be kept as explicit attributes of values. Type checking consists of generating errors for functions requiring certain values to perform their operations. For example, car and cdr require their operands to be lists: (car 2) generates an error. Types can be checked explicitly by the programmer, however, using predefined test functions.
Types in Scheme include lists, symbols, atoms, and numbers. Predefined test functions include `number?` and `symbol?`. (Such test functions are called **predicates** and always end in a question mark.)

**Example 3**

Ada is a strongly typed language, and all type errors cause compilation error messages. However, even in Ada, certain errors, like range errors in array subscripting, cannot be caught prior to execution, since the value of a subscript is not in general known until runtime. However, the Ada standard guarantees that all such errors will cause exceptions and, if these exceptions are not caught and handled by the program itself, program termination. Typically, such runtime type-checking errors result in the raising of the predefined exception `Constraint_Error`.

An essential part of type checking is **type inference**, where the types of expressions are inferred from the types of their subexpressions. Type-checking rules (that is, when constructs are type correct) and type inference rules are often intermingled. For example, an expression `e1 + e2` might be declared type correct if `e1` and `e2` have the same type, and that type has a "1" operation (type checking), and the result type of the expression is the type of `e1` and `e2` (type inference). This is the rule in Ada, for example. In other languages this rule may be softened to include cases where the type of one subexpression is automatically convertible to the type of the other expression.

As another example of a type-checking rule, in a function call, the types of the actual parameters or arguments must match the types of the formal parameters (type checking), and the result type of the call is the result type of the function (type inference).

Type-checking and type inference rules have a close interaction with the type equivalence algorithm. For example, the C declaration

```c
void p ( struct { int i; double r; } x ){
    ...
}
```

is an error under C's type equivalence algorithm, since no actual parameter can have the type of the formal parameter `x`, and so a type mismatch will be declared on every call to `p`. As a result, a C compiler will usually issue a warning here (although, strictly speaking, this is legal C). Similar situations occur in Pascal and Ada.

The process of type inference and type checking in statically typed languages is aided by explicit declarations of the types of variables, functions, and other objects. For example, if `x` and `y` are variables, the correctness and type of the expression `x + y` is difficult to determine prior to execution unless the types of `x` and `y` have been explicitly stated in a declaration. However, explicit type declarations are not an absolute requirement for static typing: The languages ML and Haskell perform static type checking but do not require types to be declared. Instead, types are inferred from context using an inference mechanism that is more powerful than what we have described (it will be described in Section 8.8).

Type inference and correctness rules are often one of the most complex parts of the semantics of a language. Nonorthogonalities are hard to avoid in imperative languages such as C. In the remainder of this section, we will discuss major issues and problems in the rules of a type system.

---

1 This has been raised to the level of an error in C++.