A Simple Mechanism for Type Security Across Compilation Units

MICHAEL L. SCOTT AND RAPHAEL A. FINKEL

Abstract—**A simple technique detects structural type clashes across compilation units with an arbitrarily high degree of confidence. The type of each external object is described in canonical form. A hash function compresses the description into a short code. If the code is embedded in a symbol-table name, then consistency can be checked by an ordinary linker. For distributed programs, run-time checking of message types can be performed with very little overhead.**

I. Introduction

A type-checking mechanism for separate compilation must strike a difficult balance between conservatism and convenience. On the one hand, it should prevent the use of compilation units that make incompatible assumptions about their interface. On the other hand, it should cause as few unnecessary recompilations as possible. When definitions change, an ideal mechanism would recompile all and only those pieces of a program that would otherwise malfunction. Approaching this ideal has proven surprisingly difficult, enough so that many programming systems provide no checking whatsoever.

We believe that a simple and easy-to-use type-checking mechanism for separate compilation is extremely important. We are particularly interested in a mechanism that can be extended to provide checking for messages exchanged between the separately loaded modules of a distributed program. We describe a technique that achieves simplicity and efficiency at the expense of an arbitrarily small probability of failure.

For the semantics of types, we adopt the rules of *structural* type equivalence [3, p. 92]. The alternative, name equivalence, requires the compiler to maintain a global name space for types. Our interest in distributed programs makes such an approach impractical:

1) A global name space requires a substantial amount of bookkeeping, even on a single machine. For a distributed language, information must be kept consistent on every node at which processes may be created. While the task is certainly not impossible, the relative scarcity of compilers that enforce name equivalence across compilation units suggests that it is not trivial either.

2) Compilers that do enforce name equivalence across compilation units usually do so by affixing time stamps to *files* of declarations. A change or addition to one declaration in a file *appears* to modify the others. A global name space for distributed programs can be expected to devote a file to the interface for each distributed resource. Mechanisms can be devised to allow simple *extensions* to an interface, but certain enhancements will inevitably invalidate all the users of a resource. In a sequential program, enhancements to one compilation unit may force the unnecessary recompilation of others. In a distributed system, enhancements to a process like the file server may force the recompilation of every program in existence.

Structural type equivalence has been used with separate compilation in a number of existing compilers [1], [5], [6]. Typically, each type declaration is converted to canonical form and is placed in the symbol table of each object file that imports or exports an object of that type. A special-purpose linker is required to guarantee that importing and exporting files contain identical canonical forms. The type information itself consumes a considerable amount of space. Comparing it byte for byte takes time. That time may be an acceptable burden in separate compilation since checks are performed at link time, but it becomes unacceptable for run-time checking in a message-passing system.

II. The Mechanism

In a note on type checking with low-level linkers [4], Hamlet credited one of his referees with the idea of using a hash function to compress the descriptions of types. No details were provided, however, and the idea appears to have lain unnoticed ever since. We arrived upon it independently nearly a decade later, and only discovered the earlier reference in the process of a literature search.

We have found the hashing of types to be an eminently practical technique. By admitting the (very slight) possibility of an undetected error, we eliminate the need for a special linker, reduce the size of object files, and allow efficient run-time message checking.

In each object file, the compiler associates a short (one- or two-word) hash code with each external object. The code for a variable depends on the canonical representation of its type. The code for a procedure depends on the types and modes (but not the names) of its parameters. The name of an external object can be formed by concatenating the name provided by the user with a character-string representation of the hash code. Type clashes between exporters and importers of an object result in "missing symbol" messages from the linker. If identifiers are limited in length, then the compiler can leave the names of objects unchanged, but can generate for each an additional symbol that encodes both the name and the type of the object. Exporting modules can "define" the extra symbols and importing modules can declare them "undefined."

In contrast to schemes that employ a special-purpose linker, our technique requires no knowlege of load-file formats or other operating-system-specific details. It may require manual intervention when clashes are detected, but this has not proven to be a serious problem in practice. We rely upon programming conventions (such as shared declaration files) to prevent the vast majority of clashes. We use the standard Unix[1] *Make* utility [2] to automate recompilation when declarations change. *Make* bases its decision on overly conservative time-stamp rules, much like those described for name equivalence in the Introduction. We can afford, however, to run the utility with incomplete rules, and to override those rules at will since the type-checking mechanism catches our mistakes.

Our hashing technique extends readily to message passing in distributed programs where applications are linked and loaded in separate pieces, and processes that need to cooperate are written at different times. For a dedicated circuit, a sender and receiver can exchange hash codes when their connection is established. Alternatively, they can exchange codes with each individual message. In either case, the extra overhead required to check for type con-

Manuscript received October 31, 1985; revised February 28, 1986. This work was supported in part by NSF Grant MCS-8105904, by ARPA Contract N0014/82/C/2087, and by a Bell Laboratories Doctoral Scholarship.

M. L. Scott is with the Department of Computer Science, University of Rochester, Rochester, NY 14627.

R. A. Finkel is with the Department of Computer Sciences, University of Wisconsin, Madison, WI 53706.

IEEE Log Number 8822012.

[1]Unix is a registerered trademark of AT&T Bell Laboratories.

sistency will be insignificant in comparison to the total cost of communication.

III. A Convenient Canonical Form and Hash Function

We have used hashing to check types for separate compilation and message passing in an experimental distributed programming language [8]. The data types in our language are similar to those of Pascal, except that there are no pointers. Pointers complicate matters; they are discussed in a subsequent section.

Our hash function is defined on strings of symbols. The symbol set includes the letters, the digits, and the underscore. We use a single letter to represent each of the type constructors **array** (*a*), **Boolean** (*b*), **case** (*u*), **character** (*c*), **const** (*k*), **end** (*f*), **enumeration** (*e*), **integer** (*i*), **link** (*l*), **record** (*r*), **set** (*s*), **subrange** (*n*), **to** (*t*), **var** (*v*), and **yields** (*y*). We use digits to represent size values. We obtain a canonical representation for a type by expanding the subparts of the type recursively. We have attempted to keep the notation as terse as possible.

For example, consider the following types:

```
A = 1..10;
B = record
        i, j : integer;
    end;
C = array [A] of B;
```

The canonical form for *A* is **"subrange integer 1 to 10,"** which we abbreviated **"ni1t10."** The string for *B* is **"riif."** The string for *C* is **"ani1t10riif."** The names of record fields are not significant.

Our message types are determined by the modes and types of the parameters of a remote operation, called an **entry.** Unlike a regular subroutine, an entry has disjoint sets of *in* and *out* parameters. To compute the canonical string for a function procedure, or entry, we concatenate the strings for the parameter types, prefixing each type with an optional mode. For example, if *foo* is a Boolean function that takes a value parameter of type character and a reference parameter of type integer, then the canonical string for foo is **"cviyb."** Similarly, if *bar* is an entry that takes one parameter of type *C* and returns two results of types integer and *B*, then the type string for bar is **"ani1t10riifyiriif."**[2]

In actuality, there is no need to compute explicit canonical forms. Our hash function treats a string of symbols as an integer base *N* where *N* is the size of the symbol set. It calculates the integer's residue modulo *p* where *p* is a very large prime. Stated precisely, let $< a > = a_{n-1}a_{n-2}a_{n-3} \cdots a_0$ be a string of symbols. Then

$$\text{hash}(< a >) =_{def} \left(\sum_{i=0}^{n-1} N^i \text{ord}(a_i) \right) \textbf{ mod } p.$$

If $< a >$ is the canonical description of a type *A*, we say

$$hashval(A) = hash(< a >) \text{ and } hashlen(A) = n.$$

In our implementation, *N* is 37 and *p* is $2^{32} - 5 = 4294967291$. The digits '0'-'9' have values 1-10. The underscore has value 11. The letters 'a'-'z' have values 12-37. No symbol has value 0 since prepending a zero-value symbol to a string would not change its hash code. The lack of a zero-value symbol allows us to use *N* for the value of 'z' without introducing ambiguity.

During compilation, we maintain two values for each type the program defines: the hash code and length of the type's canonical form. When a new type is defined in terms of existing ones, we can compute the new hash code and length from the stored information for the existing types. In the types defined above, we would like the hash code for *C* to be the same as the hash code for

```
C' = array [1..10] of record
         i, j : integer;
     end;
```

This is precisely the result we obtain by letting

$$hashval(C) = [a \times N^{hashlen(A)} + hashval(A)] \times N^{hashlen(B)} + hashval(B),$$

$$hashlen(C) = 1 + hashlen(A) + hashlen(B),$$

where ***a*** is the value of the symbol **array** as an *N*-ary digit.

All arithmetic is carried out in the ring of integers mod *p*. For our example types, the hash codes are as follows:

Type	Hash Code	String
A	1771225965	**ni1t10**
B	1497074	**riif**
C, *C'*	1948320452	**ani1t10riif**
foo	27938528	**cviyb**
bar	4056336255	**ani1t10riifyiriif**

IV. The Problem with Pointers

The technique just described must be modified to accept pointers. The problem is that forward references are needed to define circular structures. When a given type is first encountered, we may not know the nature of its constituent parts. We can still derive a canonical description and hash code for each type, but we cannot do it incrementally the way we could above.

Given a set of interrelated types, it is not difficult to determine which are structurally distinct and which are equivalent [7]. For purposes of type checking, symbol table entries for equivalent types can be coalesced. We can then use the string notation above, augmented with backpointers, to construct canonical descriptions for the types that remain. We expand each type declaration recursively until we encounter a cycle. We then insert a backpointer to the point where the cycle began. For example, the type

```
sequence = record
               item : integer;
               next : ^sequence;
           end;
```

might be represented by **"record integer pointer −3 end,"** abbreviated **"rip3f."**

[2]To demultiplex messages in a receiving process, we compute a second hash code from the *name* of the entry—hence the need in our symbol set for all the letters and the underscore.

References

[1] A. Celentano, P. D. Vigna, C. Ghezzi, and D. Mandrioli, "Separate compilation and partial specification in Pascal," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 320-328, July 1980.

[2] S. I. Feldman, "Make—A program or maintaining computer programs," *Software—Practice and Experience*, vol. 9, pp. 255-265, 1979.

[3] C. Ghezzi and M. Jazayeri, *Programming Language Concepts*. New York: Wiley, 1982.

[4] R. G. Hamlet, "High-level binding with low-level linkers," *Commun. ACM*, vol. 19, pp. 642-644, Nov. 1976.

[5] R. B. Kieburtz, W. Barabash, and C. R. Hill, "A type-checking program linkage system for Pascal," in *Proc. 3rd Int. Conf. Software Eng.*, May 1978, pp. 23-28.

[6] W. Koch and C. Oeters, "The Berlin ALGOL 68 implementation," in *Proc. Strathclyde ALGOL 68 Conf.*, Mar. 1977, pp. 102-108; also, *ACM SIGPLAN Notices*, vol. 12, June 1977.

[7] J. Kral, "The equivalence of mode and the equivalence of finite automata," *ALGOL Bull.*, vol 35, pp. 34-35, Mar. 1973.

[8] M. L. Scott and R. A. Finkel, "LYNX: A dynamic distributed programming language," in *Proc. 1984 Int. Conf. Parallel Processing*, Aug. 1984, pp. 395-401.