A SIMPLE MECHANISM FOR TYPE SECURITY ACROSS COMPILATION UNITS

by

Michael L. Scott
Raphael A. Finkel

Computer Sciences Technical Report #541

June 1984

A Simple Mechanism for Type Security Across Compilation Units

Michael L. Scott
Raphael A. Finkel

Department of Computer Sciences
University of Wisconsin - Madison
1210 W. Dayton
Madison, WI 53706

May 1984

ABSTRACT

This note describes a simple mechanism for checking structural type equivalence across compilation units. The type of each external object is described in canonical form. A hash function compresses the description into a one- or two-word code. Type checking is then performed by any standard linker, without modifications. For distributed programs, type checking can be also performed at run time, with minimal overhead.

1. Introduction

A type-checking mechanism for separate compilation must strike a difficult balance between correctness and conservatism. 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 a small but non-zero probability of failure.

2. Background

There are two principal approaches to defining type equivalence: **name equivalence** and **structural equivalence** [2]. Each has its own advantages and disadvantages [11]. Each poses its own difficulties for separate compilation.

In a language using name equivalence, each object is associated with a reference to the (lexical) declaration of its type. Each declaration defines a separate type, unique over space and time. Declarations never change; they are replaced by new and different versions, defining new and different types.

A type-checking mechanism for name equivalence implements a dependency DAG in which each program fragment points to the declarations used in its compilation. Each declaration may point to others, on which it in turn depends. Typical implementations [9, 12, 13] allow separate

This work was supported in part by NSF grant number MCS-8105904, by Arpa contract number N0014/82/C/2087, and by a Bell Telephone Laboratories Doctoral Scholarship.

compilation units to share *files* of declarations. A declaration file must be compiled before any compilation unit that uses it. The compiled version contains a time stamp. Each object file names the compiled versions of the declaration files that were used in its own compilation. For distributed programs, the naming scheme must work across machines, so that copies of declaration files at separate sites appear to be the same.

The bookkeeping required for enforcing name equivalence is simplified considerably by grouping declarations together in files. Unfortunately, that grouping also leads to an overly conservative mechanism. In effect, even the smallest change to a declaration file creates new versions of all the types the file contains. Any program unit that uses any of those types must then be recompiled. The problem is especially severe in programming systems like UW-Pascal [8], where all definitions are contained in a single **environment** file. A mechanism for *extending* an existing environment without invalidating its previous contents helps to some extent.

Many of the problems just described disappear if we adopt the more liberal rules of structural type equivalence. These rules define a type as a particular arrangement of simpler types. Each type can be declared lexically any number of times. The type-checking mechanism guarantees that interacting compilation units contain matching descriptions of the structure of their interface. As a matter of convenience, it may still be desirable to share files of declarations, but the type checking mechanism no longer depends on policing that sharing. All that matters is the *content* of declarations, and that content is checked for each individual type.

Structural type equivalence has been used with separate compilation in a number of existing compilers [1,5,6]. It introduces its own implementation problems. Each type declaration is put in canonical form and is inserted 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. We can afford that time for relatively infrequent runs of the linker, but we cannot afford it for run-time checking on messages.

3. The mechanism

In a note on type checking with low-level linkers [3], Richard Hamlet credits one of his referees with the idea of using a hash function to compress the descriptions of types. We elaborate on this suggestion. We use structural equivalence for type checking across compilation units, but we do not insist on absolute security. By admitting the possibility that occasional type errors will go undetected, 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 includes a one- or two-word hash code for each external object. The code for a variable depends on its name and on the canonical description of its type. The code for a procedure depends on its name and on the type and modes (but not the names) of its parameters. Hash codes can be treated as ordinary symbols by any standard linker. Exporting modules ``define'' the codes and importing modules declare them ``undefined.'' A utility program can translate ``missing symbol'' errors from the linker into specific, useful type-clash messages. Such a utility is preferable to a special-purpose linker because it requires no knowledge of load-file formats or other operating system-specific details.

The hashing technique extends readily to distributed programs that are linked and loaded in several pieces. In a circuit-switched network a sender and receiver can exchange hash codes when their circuit is created. In a packet-switched network they can exchange hash codes with each message. In either case, the extra overhead of checking for type consistency will be insignificant in comparison with the typical cost of communication.

4. A convenient canonical form and hash function

We have used hashing as a means of message type checking in an experimental distributed programming language [10]. The data types in our language are similar to those of Pascal [4], 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. Our symbol set includes the set of digits, as well as the symbols **array, Boolean, case, character, end, enumeration, integer, operation, record, return, set, subrange,** and **to**. Our messages are used to invoke remote operations. Logically, we obtain canonical forms by expanding the subparts of each type declaration recursively and by using an abbreviated syntax to express the results in as short a string as possible. The hash code of a remote operation is computed from the concatenation of the operation's name, the modes of its parameters, and the canonical descriptions of their types.

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. 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. Suppose, for example, we are given the following declarations:

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

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

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

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

where $\mathbf{a}$ is the value of the symbol "array" as an N-ary digit. All arithmetic is carried out in the ring of integers mod p. Further details are contained in an appendix.

5. The problem with pointers

When pointers are present the above technique breaks down. The problem stems from the need for forward references in defining 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 equivalent [7]. 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 the string "**record integer pointer -3 end**."

6. Conclusion

Type checking across compilation units has always been a major nuisance. By requiring only structural type equivalence for external references, rather than name equivalence, we can reduce the amount of work involved considerably. By hashing the descriptions of types we can summarize all the information about a compilation unit's connections to the outside world in a reasonable number of bits. Type checking can be performed at link time or run time without worrying about which units depend on which files of declarations and what was modified when. An arbitrarily high degree of security can be obtained by increasing the size of the hash-function range.

References

[1] Celentano, A., P. D. Vigna, C. Ghezzi, and D. Mandrioli, "Separate Compilation and Partial Specification in Pascal," *IEEE Transactions on Software Engineering* SE-6:4 (July 1980), pp. 320-328.

[2] Ghezzi, C. and M. Jazayeri, *Programming Language Concepts*, John Wiley and Sons, New York (1982).

[3] Hamlet, R. G., "High-Level Binding with Low-Level Linkers," *CACM* 19:11 (November 1976), pp. 642-644.

[4] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, Lecture Notes in Computer Science #18, Springer-Verlag, Berlin (1974).

[5] Kieburtz, R. B., W. Barabash, and C. R. Hill, "A Type-checking Program Linkage System for Pascal," *Proceedings, Third International Conference on Software Engineering* (10-12 May 1978), pp. 23-28.

[6] Koch, W. and C. Oeters, "The Berlin ALGOL 68 Implementation," *Proceedings of the Strathclyde ALGOL 68 Conference* (29-31 March 1977). In *ACM SIGPLAN Notices* 12:6 (June 1977).

[7] Kral, J., "The Equivalence of Modes and the Equivalence of Finite Automata," *ALGOL Bulletin* 35 (March 1973), pp. 34-35.

[8] LeBlanc, R. J. and C. N. Fischer, "On Implementing Separate Compilation in Block-Structured Languages," *Proceedings of the SIGPLAN Symposium on Compiler Construction* (6-10 August 1979), pp. 139-143. In *ACM SIGPLAN Notices* 14:8 (August 1979).

[9] Mitchell, J. G., W. Maybury, and R. Sweet, "Mesa Language Manual, version 5.0," CSL-79-3, Xerox Palo Alto Research Center (April 1979).

[10] Scott, M. L. and R. A. Finkel, "LYNX: A Dynamic Distributed Programming Language," To appear in the *1984 International Conference on Parallel Processing* (August 21-24, 1984).

[11] Tennent, R. D., "Another Look at Type Compatibility in Pascal," *Software-Practice and Experience* 8 (1978), pp. 429-437.

[12] United States Department of Defense, "Reference Manual for the Ada Programming Language," (ANSI/MIL-STD-1815A-1983) (17 February 1983).

[13] Wirth, N., "Modula-2," Report 36, Institut für Informatik, ETH Zurich (1980).

Appendix

Let $<a> = a_{n-1} a_{n-2} a_{n-3} \cdots a_0$ be a string of symbols. Then

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

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

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

In our implementation, N is 36 and p is $2^{32} - 3 = 4294967293$. We represent every symbol by a letter or digit. The digits '0'-'9' have values 1-10. The letters 'a'-'z' have values 11 through 36. 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.

Using **a**, **e**, **i**, **n**, **o**, **r**, **s**, and **t** for the symbols **array**, **end**, **integer**, **return**, **operation**, **record**, **subrange**, and **to**, respectively, we have the following hash codes for the types A, B, C, and C' in section 4:

type	**hash code**	**string**
A	1785564073	si1t10
B	1331691	riie
C, C′	2112304967	asi1t10riie

The hash codes of remote operations are computed similarly. Suppose *foo* is an operation that takes one argument of type C and returns a result of type B. The type string for foo is "**oasi1t10riienriieefoo**;" its hash code is 3966191234.

The syntax for canonical descriptions is quite terse. The intent is to use as few symbols as possible while avoiding ambiguity. The names of record fields are not significant.