International Conference on
Computer Systems and Education, IISc, Bangalore, 1994

Type-safe Composition of Object Modules*

Guruduth Banavar, Gary Lindstrom, Douglas Orr
Department of Computer Science, University of Utah
Salt Lake City, Utah 84112 USA

Abstract

We describe a facility that enables routine type-checking during the linkage of external declarations and definitions of separately compiled programs in ANSI C. The primary advantage of our server-style type-checked linkage facility is the ability to program the composition of object modules via a suite of strongly typed module combination operators. Such programmability enables one to easily incorporate programmer-defined data format conversion stubs at link-time. In addition, our linkage facility is able to automatically generate safe coercion stubs for compatible encapsulated data.

*This research was sponsored by the Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Advanced Research Projects Agency or the US Government. Contact author: G. Banavar, Computer Science - 3190 MEB, University of Utah, Salt Lake City, UT 84112 USA, e-mail *banavar@cs.utah.edu*, phone +1-801-581-8378, fax +1-801-581-5843.

1 Introduction

It is widely agreed that strong typing increases the reliability and efficiency of software. However, compilers for statically typed languages such as C and C++ in traditional non-integrated programming environments guarantee *complete* type-safety only within a compilation unit, but not *across* such units*. Longstanding and widely available linkers compose separately compiled units by matching symbols purely by name equivalence with no regard to their types. Such "common denominator" linkers accommodate object modules from various source languages by simply ignoring the static semantics of the language. Moreover, commonly used object file formats are not designed to incorporate source language type information in an easily accessible manner.

In this paper, we present a technique to perform type checking of object modules as a routine link-time activity. Our technique is characterized by (i) the design of specific language type systems into a system-wide linker, (ii) programmed link-time control over individual symbols of object modules, and (iii)

*C++ style name-mangling does not accomplish complete type-safety across compilation units; see Section 6.

utilization of standard debugging information generated by compilers for type checking. We describe in detail the realization of these steps for ANSI C.

A crucial enabler for this facility is the ability to resolve inconsistencies among compiled object modules at link time. The existence of link time type errors does not mean that program source files need to be modified and recompiled, as this may not be possible for pre-compiled libraries. Programmer control for correcting link time type errors is provided via the already existing programming facilities of OMOS [17], our dynamic linker. For instance, consider the case where the type of a declaration in one translation unit does not match with a definition in another. This can usually be fixed by (i) uniformly renaming the declaration and its uses to match the actually intended definition name, or (ii) in the case when the names match but the types do not, by introducing a new declaration to match the definition, and binding the renamed original declaration with a type conversion function. Our linkage facility easily supports such transformations. If a type error cannot be corrected with such simple transformations on object modules, it might indicate a more serious error in the design of the modules involved.

Our link-time type-checking facility permits us to adapt and utilize the full expressive power of language type systems to better suit modern persistent, distributed and heterogeneous environments. For example, *structural* typing can be applied to languages such as ANSI C with *name-based* typing. Pure name-based typing becomes a problem in persistent and distributed environments, where data and types could migrate outside the program in which they were originally created [1], and lead to matching of names that may or may not have the same programmer-intended meaning. This argues for structural matching of aggregate types similar to Modula-3 [15], using member order and type significance along with names.

Furthermore, our programmable linkage facility enables the incorporation of automatic and user-defined conversion routines for encapsulated data. For automatic conversion, we postulate safe adaptability rules for converting built-in data types using the language definition in conjunction with the characteristics of particular hardware platforms. We then utilize these rules to automatically generate data conversion "stubs" at link time. More importantly, programmer defined conversion stubs can also be easily incorporated at link time. This opens up the possibility of programmer-controlled data evolution and conversion across heterogeneous data formats, *e.g.* those arising from different languages, hardware architectures, *etc.*.

We provide the ability to support a variety of type systems by designing our type-checking facility as an extension of an object-oriented *framework* [2]. The O-O framework contains generic type system related abstractions such as *named types*, *function types*, *record types*, *etc.* that are specialized via inheritance to implement the type domain of specific languages.

In the following sections, we describe in detail the type-checking of object modules generated by compiling ANSI C programs. Section 2 introduces our notion of modules and interfaces, Section 3 briefly describes our object server OMOS, and Section 4 discusses the essential aspects of the type system of ANSI C. We then give some implementation details, discuss related work and conclude.

2 Object Modules and their Interfaces

We refer to an ANSI C program source or object file as a *module*, consisting of a set of *attributes* with no order significance. An attribute is either a file-level declaration (a name with an associated type, *e.g.* `extern int i;`), or a file-level definition (a name with a data, storage or function *binding*). Type definitions (e.g. `struct` definitions, and `typedef`'s in C) are not attributes of a module. The *interface* of a module consists of <*name, type, declared_or_defined*> tuples of the *attributes* of the module. In the context of type-checking object module interfaces, attributes *match* if they have the same name. There cannot be matching attributes within a single interface†, and attributes that match across interfaces must be type compatible. The notion of type compatibility depends on the particular module combination operation being performed, and is informally described below.

Our linker is based upon a formal model of modules proposed in [5], achieving a fine level of control over individual attributes of object modules. Briefly, object modules are combined via a suite of module combination *operators* that were originally conceived to describe the many facets of inheritance in object-oriented programming. Figure 1 gives the primary operators, their informal semantics and type rules. These operators provide control over aspects of visibility, sharing, and rebindability of individual attributes of modules. The power that this model lends to object module linkage is briefly given in Section 3, and is described in more detail in [17], where the original implementation of the type-less OMOS linker is described. The current effort incorporates the rules of the strongly typed module model and illustrates some of its applications.

The semantics of common linkage is embodied in the module operator `merge`. For a simple example of the use of this module operator, consider Figure 2. In this figure, the compiled module `O1` provides a definition of function `f`. Consider the case where a programmer creates and compiles module `O2` with the intention of using `O1`'s `f` definition by performing `O1 merge O2`, but makes the incorrect presumption that `f` returns an `int`. If `merge` were untyped (as it is in common linkage), `O1 merge O2` would have been legal; however, it does not typecheck in our linker since the interfaces of `O1` and `O2` are not type compatible for a `merge` operation.

Let us say that the programmer of `O2` discovers during linkage that `f` returns the desired `int` value as a component of the returned structure. Traditionally, in order to make `O1` and `O2` compatible, the programmer would modify the source code of either module extensively, if it were available, and recompile. This, of course, could adversely affect combination of the modified module with yet other modules. Alternatively, in our model of flexible link-time module adaptation, `O2` can be adapted to get the desired effect by constructing a "stub" module `O3`. `O3` consists of a new declaration that matches `f`'s definition, and a stub function that extracts the desired value from the structure returned by `f`. With this, a modified version of `O2` is obtained with the module expression `(O2 rename f f_stub) merge O3`, which can then be `merge`'ed with `O1` to get the originally desired effect.

3 The OMOS Linker

In this section, we describe our linkage facility, the Object Meta-Object Server OMOS[17].

The OMOS linker/loader is designed to provide a dynamic linking and loading facility for client programs via the use of module combination and instantiation. OMOS implements a persistent hierarchical namespace — much like the UNIX directory hierarchy — whose leaf nodes are either object modules (`.o` files) or *meta-objects*. Meta-objects

†In order to model languages that support user-defined overloading, *e.g.* C++, our model can be extended to include an *ordinal* value in the tuple, which is also significant for attribute matching.

Operator	Semantics	Typing
`M1 merge M2`	Combine `M1` and `M2`.	Matching definitions disallowed; a definition must be a subtype of its matching declaration.
`M1 override M2`	Merge, but resolve matches in favor of right operand.	Same as merge, except right-operand definition must be a subtype of matching left opnd. definition.
`M restrict L`	Make `L` undefined.	`L` must be defined.
`M freeze L`	Make references to `L` static.	`L` must be defined.
`M hide L`	Make attribute `L` "private."	`L` must be defined.
`M rename L1 L2`	Rename `L1` & its uses to `L2`.	`L1` must be defined; `L2` must not exist.
`M copyas L1 L2`	Copy attribute named `L1` to `L2`.	`L1` must be defined; `L2` must not exist.

Figure 1: Informal Semantics and Typing of Module Operators

```
/* Module O1: */
struct S {
  int x;
  /* ... */
 }

struct S f () {
  /* ... */
}
/* ... */
```

```
/* Module O2: */

extern int f ();

void bar () {
  int x = foo ();
}

 /* ... */
```

```
/* Module O3: */
struct S {
   int x;
   /* ... */
}
extern struct S f ();
int f_stub () {
   return f().x;
}
```

Modules `O1` and `O2` are composed with the expression: `(O2 rename f f_stub) merge O3 merge O1`

Figure 2: Linkage Adaptation

are named placeholders for modules that are specified by module combination expressions. OMOS essentially provides a level of indirection between a named OMOS entity (a module) and its actual implementation (a module *instance*) that is loaded into a client. Clients may directly load named module implementations or generate new modules by combining or modifying existing ones. This facility is used as the basis for system program execution and shared libraries[16], as well as dynamic loading of simple modules.

Expressions specifying module combination are encoded in a scripting language with a LISP-like syntax. These expressions consist primarily of operations for manipulating modules and module namespaces, such as those shown in Figure 1. Additionally, OMOS supports operations for constructing an object module given program source code, and for specializing the implementation of a given module (*e.g.* library vs. ordinary module) [16], among others. The operands in module expressions may be executable code or data fragments, other module expressions, or other named meta-objects.

Since OMOS is an active entity (a server), it is capable of performing sophisticated module manipulations on each instantiation of a module. Evaluation of a module expression could potentially produce different results each time. Some OMOS operations such as those used to implement program monitoring and reordering [18] enact program transformations using operations on module expressions.

For example, monitoring a program using OMOS might involve extracting and transforming the expression that generates the program so that each defined procedure is transparently *wrapped* with an outer routine that monitors entry to and exit from the procedure. Figure 3 shows the module operations used to "wrap" the procedure `f` in module `O1` with the automatically generated routine found in `O2`. Note that this illustrates adaptation of the "service provider" module, while Figure 2 showed client module adaptation.

This process of wrapping procedures is enhanced by the availability of module type information. The wrapper procedure is constructed with a signature identical to that of the wrapped procedure; simple language constructs can be used to propagate the caller's arguments to the wrapped routine. If type information was not available (or in cases such as `printf` where the the routine is defined to take a variable number of arguments) it would be necessary to use a machine-dependent wrapper that could preserve and pass along the call frame without knowledge of its contents.

While OMOS is capable of performing sophisticated manipulations on each invocation, it caches the results of most operations to avoid re-doing work unnecessarily. The practice of combining a caching linker with the system object loader gives OMOS the flexibility to change implementations as it deems necessary, *e.g.* to reflect an updated implementation of a shared module across all its clients [16].

4 C's Type System

This concludes the general discussion of linkage via module manipulation.

In order to ascertain the type-safety of modules being combined, the module type rules (shown informally in Figure 1) built into our linker requires knowledge of the type system (type domain, type equivalence and subtyping) of the base language ANSI C. This section describes the relevant type system of ANSI C (type domain and type equivalence) [11], and enhancements made to it for type-checking across compilation units (structural typing, and subsumption).

The type domain of ANSI C consists of (i) basic types (primitive types (`int`, `float`, *etc.*), and enumerated types), (ii) derived types (function types, struct and union types, array and pointer types), and (iii) `typedef`'ed names. Specifiers for these types can be augmented with type qualifiers (`const` and `volatile`) and storage class specifiers (`auto, register, static` and `extern`).

The type qualifier `volatile` concerns optimization, and is not relevant here. The qualifier `const` is explicitly dealt with in this section. The storage class specifiers `auto` and `register` are not relevant since they may only be used within functions — we are interested in file-level declarations and definitions. The storage specifier `extern` indicates an attribute *declaration* while non-extern attributes are considered to be *defined*. The storage specifier `static` for a file-level attribute gives it internal linkage, *i.e.* the attribute can be viewed as having been subjected to a `hide` module operation. Similarly, attributes that are subjected to `hide` via link-time programming can be regarded as having been converted to the `static` storage class after the fact.

C permits calls to functions that have not been declared in a module. A call to an undeclared function `f` in a module results in an implicit file-level declaration of `extern int f ()`.

4.1 Type Equivalence

Type equivalence in ANSI C within a single translation unit, and our extensions for type-checking across translation units, is given in Figure 4. The rationale for the two modifications are

1. For aggregate types (`struct`'s and `union`'s), name equivalence is too weak

```
/* Module O1: */




void g () {
    short z = f (3);
}

short f (short x) {
    /* ... */
}
```

```
/* Module O2: */
/* Automatically generated */
extern short __f (short);
extern void _log_enter (char *);
extern void _log_exit (char *);
short f (short x) {
    short v;
    _log_enter ("f");
    v = __f (x);
    _log_exit ("f");
    return v;
}
```

Module expression: `(((O1 copyas f __f) restrict f) merge O2) hide __f`

Figure 3: Wrapping a routine to monitor its execution

Type	Equivalence within a translation unit	Equivalence across translation units
Primitive type	name equivalence	same
Function type	structural, with in and out parameter types significant	same
Enum type	name equivalence	same
Structure and union type	name (tag) equivalence; tag-less types are unique	structural, with tag, member order and member names significant
Pointer type	equivalence of target types	same
Array type	equivalence of element types, and equality of array size	same
`typedef`'ed name	`typedef`'ed type	`typedef` name equivalence

Figure 4: Type equivalence in ANSI C

when applied outside of a single translation unit, as explained in the introduction. Therefore, we adopt a conservative structural typing regimen in which the names, order and types of members are also significant. We also retain the significance of aggregate tags since there could be application-specific semantic content in them.

2. For `typedef`'ed names, again, there could be application-specific semantic content in them, so we adopt strict name equivalence.

Furthermore, some type specifiers are implied by others, *e.g.* `short` implies `short int`, therefore these types are equivalent. The type qualifier `const` is significant for equivalence since it distinguishes read-only variables from read-write variables.

4.2 Subtyping

The module operators `merge` and `override` utilize subtyping rules for type-checking combination. Our base language, ANSI C, has no notion of subtypes; hence subtyping can be considered to be restricted to type equivalence. However, module composition would be more flexible if we could retroactively formulate subtyping rules consistent with the language definition.

The ANSI C language specifies safe con-

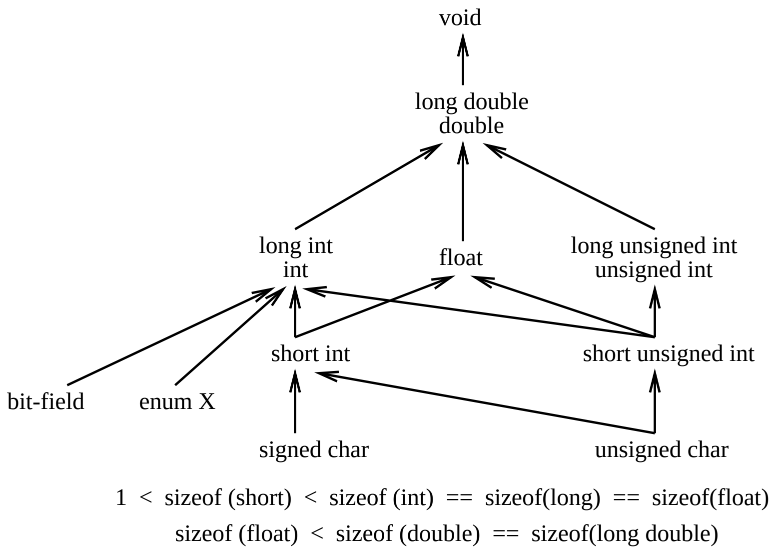

Figure 5: Subtyping of C Primitive Data Types

version rules for certain primitive arithmetic data types (*e.g.* `float` to `double`). A conversion is said to be safe if all values of one type can be represented as values of the the other without loss of precision or change in numerical value. C compilers, however, can usually be expected to support many more safe conversions than those that are defined by the language, as governed by hardware characteristics. These safe conversion rules can be thought of as *subsumption* rules, which in turn provide the basis for formulating subtype rules for primitive arithmetic types. Figure 5 shows the data type sizes and a partial order of subtypes for the HP series 9000 machines (300s and 700s). For instance, a value of type `short` can be safely coerced into a value of type `float` on this platform without loss of precision or change in numerical value. We might ask if the above rules can be exploited during type-checking of attributes across translation units.

Consider file-level variable declarations. Variables can be used as evaluators (*i.e.* expressions that return values) and as acceptors (*i.e.* expressions that receive values) in different contexts. Expressions which are evaluators can only be replaced with expressions whose types are subtypes of the original, while expressions which are acceptors can only be replaced by expressions whose types are supertypes of the original [6]. As a result, subtyping of variables is always restricted to type equivalence.

Consider file-level read-only (*i.e.* `const`) variables. Subtyping involving the type qualifier `const` can be described as follows: if a non-const type s is a subtype of a non-const type t, then `const` s is a subtype of `const` t, s is a subtype of `const` t, but `const` s is *not* a subtype of t. So, for example, can a declaration `extern const float x` in one translation unit be considered subsumed by a definition `short x` in another?

Unfortunately, this is not the case, since size and layout formats for various primitive data types are almost certainly incompatible. Moreover, in certain cases, *e.g.* `enum` types, compilers usually optimize layout by packing, hence the fact that an `enum` type is really an `int` cannot be utilized. Within the *same* translation unit, however, such subsumption rules can be applied since the compiler has complete knowledge of layout and usage and hence it can generate appropriate conversion and access code.

Similar arguments hold for subtyping constant user-defined aggregate data types (`struct` and `union`) across translation units. For example, a `struct` of two `short`s cannot be considered to be a subtype of a `const struct` of two `const float`s even though `short` is a subtype of `const float`. Furthermore, C `union`s are not discriminated, and member access is not type-checked at runtime. For example, a `union` with one `short` component cannot be read-only accessed by a supertype, a `union` with a `const short` and a `const float` component, in another translation unit, since there is no way for the supertype accessor to know at run-time if the `union` actually contains a `short` value or a `float` value. As a result, subtyping on file-level read-only variables is also restricted to type equivalence.

Arguments such as the above can be formulated to show that subtyping on pointer types is also restricted to type equivalence.

Consider subtyping of function types. Subtyping of function types is by *contravariance* [6]. That is, a function type is a subtype of another with the same number of arguments if its return type is a subtype of the latter's, and its input argument types are *supertypes* of the corresponding ones in the latter. According to this rule, one can pass a function actual parameters that are *subtypes* of the formal parameters in the function definition. For subtyping function types with an unspecified (variable) number of arguments, we require that the subtype has at most the number of explicitly specified argument types in the supertype, and that they are in the proper relationship.

However, we cannot use such a rule across translation units since in a compiled function, the amount of space allocated for the input parameters is exactly the size of the expected types, and the format is expected to be exactly as specified. All in all, and not surprisingly, *no* useful subtyping rules can be discovered in the existing C language for *direct* application in type-checking across translation units.

The crucial observation, however, is that several useful subsumption rules can be utilized for data that are *encapsulated* within functions, *if* "stubs" that perform the appropriate coercion between data-types can be inserted between combined modules at link time. This is feasible since such stub functions are themselves compiled and hence they can utilize data format conversion knowledge that a compiler uses within a translation unit. Applying this stub technique to global *data*, however, is not feasible since it involves initializing global variables with non-constant values, which is illegal in ANSI C.

Function types lend themselves particularly well to this technique since the performance of function calls is affected much less by this indirection than the performance of data access. Moreover, it does not seem unreasonable to impose the requirement on users to encapsulate such data that they foresee will be accessed via supertypes.

Our linker automatically generates coercion stubs for functions using the primitive type conversions shown in Figure 5. For an example of type adaptation using language defined subtypes, consider Figure 6. As mentioned earlier, the type `short` is a subtype of `float`. Therefore, the definition of function `f` in module `O1` is a subtype (by contravariance) of the declaration of the function `f` in module `O2`. However, `O1` cannot be directly merged with `O2`, since in general the calling sequence for `f` might not be compatible, *e.g.* the definition of `f` might be expecting its input in a floating point register rather than an integer register. This is remedied by first combining `O2` with the automatically generated stub module `O3` that incorporates safe coercions, and then performing the desired merge, as shown in the figure.

We have also incorporated a comprehensive subtyping model including structural

```
/* Module O1: */          /* Module O2: */              /* Module O3: */
                                                        /* Automatically generated */
short f (float y) {       extern float f (short);       extern short f (float);
  /* ... */
}                         void g () {                   float f_stub (short x) {
                            float z = f (3);              return (float) f ((float) x);
                          }                             }
```

Modules `O1` and `O2` are combined with the expression:
`(((O2 rename f f_stub) merge O3) hide f_stub) merge O1`

Figure 6: Automatic Data Coercion Using Language Rules

record subtyping with member name, type and order significance, an example of which is shown in Figure 7. It should be emphasized that the above technique applies only to input and output parameters of functions, since coercion stubs can be automatically generated to account for function subtyping only.

This technique of type conversion stubs can be generalized as illustrated in Figure 8 to provide a general facility to incorporate user defined stubs at link time for arbitrary data format conversion. In the figure, module `O3` comprises user-defined stubs.

5 Implementation And Usage Details

Ideally, we would have compilers that generate object modules in a "self-describing" format, with information about the source language, the machine architecture, and the interface, all packaged within the object module in a readily accessible format. However, this is far from reality — the closest approximation is an object file that has been compiled with the debugging option[‡] `-g`, which instructs the compiler to generate type information in a standard encoded format.

Although conceptually simple, the actual process of extracting type information from the generated debugging information is technically challenging, and in our prototype involved the following steps. The GNU C compiler, `gcc`, does not generate debugging information for C `extern` symbols, since debugging is normally performed on executable files in which all external references have been resolved. To solve this, we modified the back end of `gcc` to generate debugging information for all symbols. For accessing the sections of the object file that contain debugging information (`.stab` and `.stabstr`), we use Cygnus Corporation's Binary File Descriptor (BFD) library [8], and parse the "stabs" format debug strings [13] using a yacc/lex generated parser.

We are implementing an O-O framework in C++ [2] that embodies the formal module model that was briefly described in Section 2. The abstractions (classes) in the framework implement the type rules discussed in the previous section. For instance, the framework class `CPrimType` implements the partial order of primitive types introduced before. A framework class called `Interface` implements the type rules for module operations such as `merge`. The parser mentioned in the previous paragraph instantiates the appro-

[‡]Object files compiled without the debugging option contain no type information, and those compiled with the debugging option contain more information than is necessary for type-checking linkage, *e.g.* types of local variables, line numbers, *etc.*.

```
/* Module O1: */

struct S {
  short x;
  float y;
}

struct S f () {
  /* ... */
}
```

```
/* Module O2: */

struct S {
  float x;
}

extern struct S f ();

void g () {
 /* ... */
}
```

```
/* Module O3: */
/* Automatically generated */
struct S1 {
    short x;
   float y;
  }
  struct S {
   float x;
  }
  extern struct S f ();
  struct S f_stub () {
     struct S1 s1;
     struct S* s = (struct S*) &s1;
     struct S ret_s;
     *s = f ();
     ret_s.x = (float) s->x;
     return ret_s;
   }
```

Modules O1 and O2 are combined with the expression:
`(((O2 rename f f_stub) merge O3) hide f_stub) merge O1`

Figure 7: Automatic Conversion of `structs` Using Structural Subtyping

```
/* Module O1: */
R1 f (T1 y) {
  /* ... */
}
```

```
/* Module O2: */
extern R2 f (T2);

void g () {
 R2 z = f (/*T2 value*/);
}
```

```
/* Module O3: */
extern R1 f (T1);

R2 f_stub (T2 x) {
    return R1_to_R2 (f (T2_to_T1(x));
}
R2 R1_to_R2 (R1 r) {
   /* ... */
}
T1 T2_to_T1 (T2 t) {
   /* ... */
}
```

Modules O1 and O2 are combined with the expression: `(O2 rename f f_stub) merge O3 merge O1`

Figure 8: Programmer-defined Data Conversion

priate classes in our O-O framework to create the interface of the object module.

For using our type-checked linkage facility, the source programs currently must be written in ANSI C, and function declarations specified using "new-style" prototypes. Furthermore, usage of header files can be minimized; explicit declarations of external functions can be provided instead. Programs that are to be type-checked at link time must be (re)compiled with our (modified) compiler using the debug (`-g`) option.

One legitimate concern is the size of object files as a result of the inclusion of debugging information. The size of object files does increase significantly due to debugging information, but this problem is exacerbated by the inclusion of huge library header files. Our solution to this problem is that given type-checking at link-time, it is not necessary to include header files in the traditional way. Instead, programs can explicitly declare prototypes for those external (library) functions that are called. A discussion of the disadvantages of header files used in the traditional manner is found below in Section 6.

6 Related Work

Integrated Development Environments (IDE's) for strongly typed languages, *e.g.* Eiffel [14], undoubtedly utilize mechanisms for type-checking separately compiled modules, since they have complete knowledge and control over source and object modules. However, our work differs from IDE's in that we provide a systemwide linkage facility that attempts to typecheck combined modules independent of language processors. Furthermore, the programmability of our linker enables "fine tuning" the compatibility of (possibly heterogeneous) object modules at link time.

Use of header files has been a longstanding attempt at type-safety of separate compilation. The *Annotated C++ Reference Manual* [10] (page 122) explains the inadequacy of header files as follows:

> "... C tried to ensure the consistency of separately compiled programs by controlling the information given to the compiler in header files. This approach works fine up to a point, but does involve extra-linguistic mechanisms, is usually error-prone, and can be costly because of the need to have other programs (in addition to the linker and the compiler) know about the detailed structure of a program."

Instead of including header files, it is clearly more modular and less error-prone to explicitly declare the expected external functionality (*e.g.* library functions), let the linker check consistency at link time, and correct inconsistencies via programming.

With the objective of enabling type-safe linkage within the constraints of existing linkers, Stroustrup [19, 10] describes a mechanism for encoding functions with the types of input arguments. However, this mechanism is inadequate for our purposes since (i) certain classes of type errors cannot be detected (page 126 of [10]) since variable types and function return types are not encoded, (ii) although it could be extended to deal with structural typing of C aggregate types, it does not scale well to arbitrarily large types, *e.g.* large `struct`s, and (iii) we want to do not only type-checking, but also useful adaptation during link-time, hence we must utilize sophisticated linker technology.

The Berkeley Pascal Compiler `pc` [9] is similar to our effort in that it employs debugging information to check type consistency across separately compiled modules. The compiler routinely generates stab-format type information into object modules, which is used by a binding phase of the compiler to check consistency before delegating the actual linking to `ld`. However, the crucial advantage with our approach is that we perform type-checking as a controlled and programmable link-time activity.

There is a plethora of literature related to stub generation[4, 12, 3, 20]. The Polygen system [7] is representative of automatic stub generation for programming in a heterogeneous environment. Polygen packages heterogeneous modules by utilizing a programmer-defined specification of their interfaces and execution environments specified in a common module language. The packaging process involves generation of client and server stubs that handle module interconnection and data type coercion dynamically. Our technique differs from Polygen in that we enable the combination of precompiled object modules by automatic extraction of interfaces and via link-time programming.

7 Ongoing Work

We are currently completing our implementation, and look forward to get more experience in using such a type-safe linkage facility. We acknowledge the shift in the traditional cycle of programming that may be required as a result of using a programmable type-checking linker. Also, automatically generating stub functions for all varieties of type compatible functions is considerably hard. For example, generating sensible stubs for function calls involving reference parameters (*i.e.* pointer parameters in C) is somewhat more difficult and is currently being worked on.

We foresee several applications for our type-safe linkage facility. In the immediate future, we plan to extend this technique to apply to O-O languages such as C++, whose type systems are significantly more complex than the simple type system of C. Furthermore, if type equivalence and subtyping rules can be established across programming languages, our facility enables multilingual programming.

Link-time type checking of module combination also opens up the possibility of more expressive type systems. The current status of static type systems for O-O languages is unable to deal with, for example, polymorphic inheritance operators which has several software engineering applications.

We are currently in the process of extending OMOS to include a small LISP interpreter to replace the special-purpose module expression language. This change will allow conditional processing of modules, definition of functions, etc. In addition, we are producing an interface to OMOS that will allow it to subsume the role of the system linker.

8 Conclusion

We have described a programmable linkage facility for separately compiled ANSI C object modules. The programming model of our linker is based on a formal notion of modules and their composition via a suite of strongly typed operators. We design the type system of ANSI C into our linker and typecheck composition by extracting the interfaces of object modules compiled with debugging information. Furthermore, we automatically generate conversion stubs for compatible encapsulated types, and permit easy incorporation of arbitrary user-defined type conversion stubs at link time. We have thus demonstrated a powerful, flexible, and type-safe linkage facility.

Acknowledgments

We are very thankful to Robert Mecklenburg and Jay Lepreau for numerous useful comments, and to Pete Hoogenboom and Jeffrey Law for sharing their knowledge of the inner workings of current compilers and linkers. The insights and support of Tim Moore, Benny Yih, and all other Mach Shared Objects project participants are also gratefully acknowledged.

References

[1] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September, 1993.

[2] Guruduth Banavar and Gary Lindstrom. A framework for module-based language processors. Computer Science Department Technical Report UUCS-93-006, University of Utah, March 5, 1993.

[3] B.N. Bershad, T.E. Anderson, E.D. Lazowska, and H.M. Levy. Lightweight remote procedure call. *Association for Computing Machinery Transactions on Computer Systems*, 8(1):37–55, February 1990.

[4] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *Association for Computing Machinery Transactions on Computer Systems*, 2(1):39–59, February 1984.

[5] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992. Technical report UUCS-92-007; 143 pp.

[6] Kim B. Bruce. A paradigmatic object-oriented programming language: Design static typing and semantics. Technical Report CS-92-01, Williams College, January 31, 1992.

[7] John R. Callahan and James M. Purtilo. A packaging system for heterogeneous execution environments. *IEEE Transactions on Software Engineering*, 17(6):626–635, June 1991.

[8] Steve Chamberlain. libbfd. Free Software Foundation, Inc. Contributed by Cygnus Support, March, 1992.

[9] 4.3 Berkeley Software Distribution. *UNIX Programmer's Supplementary Documents*. University of California, Berkeley, California 94720, April 1986.

[10] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.

[11] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988.

[12] B. Lyon. Sun remote procedure call specification. Technical report, SUN Microsystems, 1984.

[13] Julia Menapace, Jim Kingdon, and David MacKenzie. The "stabs" debug format. Free Software Foundation, Inc. Contributed by Cygnus Support, 1993.

[14] Bertrand Meyer. Eiffel, the environment, August 1989.

[15] Ed. Greg Nelson. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[16] Douglas Orr, John Bonn, Jay Lepreau, and Robert Mecklenburg. Fast and flexible shared libraries. In *Proc. USENIX Summer Conference*, pages 237–251, Cincinnati, June 1993.

[17] Douglas B. Orr and Robert W. Mecklenburg. OMOS — An object server for program execution. In *Proc. International Workshop on Object Oriented Operating Systems*, pages 200–209, Paris, September 1992. IEEE Computer Society. Also available as technical report UUCS-92-033.

[18] Douglas B. Orr, Robert W. Mecklenburg, Peter J. Hoogenboom, and Jay Lepreau. Dynamic program monitoring and transformation using the OMOS object server. In *Proceedings of the 26th Hawaii International Conference on System Sciences*, pages 232–241, January 1993. Also available as technical report UUCS-92-034.

[19] Bjarne Stroustrup. Type-safe linkage for C++. In *USENIX C++ Conference*, 1988.

[20] Satish R. Thatte. Automated synthesis of interface adapters for reusable classes. In *Symposium on Principles of Programming Languages*, January, 1994.