Translucent Sums: A Foundation for Higher-Order Module Systems

(Dissertation Summary)

Mark Lillibridge

December 12, 1996

Abstract

The ease of understanding, maintaining, and developing a large program depends crucially on how it is divided up into modules, which is constrained by the available modular programming facilities ("module system") of the programming language being used. Experience with Standard ML's module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically. Module systems with such facilities are called higher-order, by analogy with higher-order functions.

Previous higher-order module systems can be classified as either "opaque" or "transparent." Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation.

In this dissertation, I describe a novel approach that avoids these problems based on a new type-theoretic concept, the translucent sum. A translucent sum is a kind of weak sum that can optionally specify the identity of its type components. Under my approach type identities are not determined by inspecting module implementations, permitting separate compilation. By default, module operations reveal the connections between the types in their input and output modules. However, these connections can be obscured using type coercion. This controlled visibility permits data abstraction where desired without limiting the uses of higher-order features. My approach also allows modules to be treated as first-class values, a potentially valuable feature.

In order to lay out the groundwork for my new approach to designing higher-order module systems and to demonstrate its utility, I develop in complete detail a kernel system using my approach. I establish formally the theoretical properties of the system, including soundness even in the presence of side effects. I also show how the system may be effectively type checked.

1 Introduction

Many programming languages have a collection of facilities for building modular programs. These collections of facilities, called *module systems*, play an important role in programming languages, especially with regard to programming-in-the-large. This dissertation concerns a new approach to designing module systems for statically typed programming languages. The approach promises a substantially more powerful module system than those built with previous approaches while at the same time eliminating the problems associated with the previous approaches.

Traditional module systems, such as the one provided by Modula 2 [14], are *first-order*, allowing only trivial manipulations of modules. Some newer programming languages provide *higher-order* module systems. Higher-order module systems, unlike first-order ones, permit the non-trivial manipulation of modules within the language. In particular, they at least permit functions mapping modules to modules and may provide other higher-order features such as modules containing modules as subcomponents and modules as first-class values. Experience with Standard ML's module system has shown the usefulness of functions mapping modules to modules and modules with module subcomponents. For example, functions over modules permit abstract data types (ADTs) to be parameterized by other ADTs, and submodules permit modules to be organized hierarchically.

Previous higher-order module systems can be classified as either "opaque" or "transparent." Opaque systems totally obscure information about the identity of type components of modules, often resulting in overly abstract types. This loss of type identities precludes most interesting uses of higher-order features. Transparent systems, on the other hand, completely reveal type identities by inspecting module implementations, which subverts data abstraction and prevents separate compilation. My thesis is as follows:

> *By basing a module system on a new type-theoretic concept, namely a new kind of weak sum that can contain both transparent and opaque type definitions called a* translucent sum*, it is possible to obtain a higher-order module system with the advantages of both the opaque and the transparent approaches, but none of their disadvantages.*

In order to demonstrate this thesis, I design a new programming language with a higher-order module system based on translucent sums. The language I have created is a kernel system that contains only the features relevant to module system design; considerable extension would be required to make it into a real programming language. The design of this kernel system and the proofs required to establish formally its properties form the core of this dissertation. Included are a proof of the system's soundness and effective procedures for type checking its programs.

The rest of this document is organized as follows: Section 2 provides the necessary background about module systems, Section 3 explains the previous approaches to designing higher-order module systems, Section 4 describes the basic ideas behind my new approach to designing module systems, Section 5 describes what translucent sums are and how they work, Section 6 briefly summarizes the features and properties of the kernel system, Section 7 briefly surveys related work, and Section 8 summarizes the dissertation's contributions.

2 Module Systems

A *module* binds a set of values and types to names. For example, we might have

```
M = module
        val  x = 3;
        val  y = true;
        type T = int;
    end;
```

This statement creates a new module with three components, called `x`, `y`, and `T`. The components `x` and `y` are value components, which are bound to the values `3` and `true` respectively; the component `T`, by contrast, is a type component, which is bound to the type `int`. Once this module is created, it is then bound to the name `M`; this binding will allow us to refer to the new modules' components as `M.x`, `M.y`, and `M.T` later on. Although in this example, we named our new module, this is not required. Sometimes, for example, we may want to create a new module then immediately apply a function to it; naming the new module serves no purpose in this case.

Because I am dealing with statically typed languages, modules will have "types", called *interfaces*. For example, the module `M` has the following interface:

```
M : interface
       val  x:int;
       val  y:bool;
       type T;
    end;
```

This interface specifies that `M` has two value components, `x` with type `int` and `y` with type `bool`, and one type component, `T`.[1]

First-order module systems have only the trivial module facilities discussed so far: module creation, module naming, and module component extraction (`M.x`). Most traditional module systems are of this type. Examples include Ada [12], CLU [8], C [6], C++ [13], and Modula 2 [14].

Higher-order module systems, by contrast, have non-trivial module manipulation facilities. I shall be concerned in this dissertation primarily with three such facilities: *functors*, submodules, and modules as first-class values. Functors are functions mapping modules to modules. For example, we might define a rectangle ADT parameterized by a point ADT using a functor:

```
MkRect = functor(p:POINT):RECT
             module
                 type T = p.T * p.T;
                 ...
              end;
```

[1]In my dissertation I deal with a more complicated version of modules that assigns "types", called *kinds*, to types; I have simplified things here.

The advantage of doing this is that we can create several rectangle ADTs from the same code using different point ADTs. For example, if we have point ADTs `CartPoint` and `PolarPoint`, we can create a rectangle ADT based on Cartesian points with `CartRect = MkRect(CartPoint)` and a rectangle ADT based on polar points with `PolarRect = MkRect(PolarPoint)`.

Submodules are modules contained as components within other modules; they allow packaging up a series of modules into a single unit. For example, if we had several search-tree modules, we could package them up into a single module:

```
SearchTree = module
                 mod Binary   = BinarySearchTree;
                 mod RedBlack = RedBlackSearchTree;
                 mod BTree    = BTree;
             end;
```

`SearchTree`, in turn, could be included as part of a larger library algorithms module.

Modules as first-class values refers to the ability to treat modules as if they were ordinary values of the programming language. This means that anything that can be done with a ordinary value can be done with a module. Thus, in languages with modules as first-class values, modules can be passed to or returned by ordinary functions, stored in variables, or selected using conditional statements. One useful way to use this facility is to select the most efficient implementation for an ADT at runtime. For example, the best implementation for a dictionary depends on the number of items it will contain; the following code sets `Dictionary` at runtime to use an implementation that is efficient for `n` items:

```
Dictionary = if n<20 then LinkedList else HashTable;
```

The most well known programming language with a higher-order module system is Standard ML (SML) [4]. SML provides functors and submodules, but treats modules as second-class values. The extensive experience of the SML community with functors and submodules has established their value. Modules as first-class values shows promise as a valuable addition to module system toolkits, but since this facility has never been implemented together with the other higher-order facilities, experience about its value is lacking. More extensive examples of the use of these features and their value can be found in Section 1.2 of my dissertation.

3 Previous Approaches

Previous approaches to designing higher-order module systems can be classified into "opaque" and "transparent" approaches depending on how they treat module and functor boundaries.

3.1 The Opaque Approach

Under the opaque approach, module and functor boundaries are opaque, allowing no information about the identity of type components to pass through. This is exactly what we want when we build an ADT. For example, consider the following definition of an integer stack ADT:

```
IntStack = module
                type T = int list ref;
                fun  new() = new [];
                fun  push(s:T,e:int) = ...
                ...
            end;
```

The identity of the integer stack type (`T`) is visible inside the module, allowing us to implement its operations using the appropriate primitives on its representation type, but its identity is obscured outside the module: `IntStack.T` is an abstract type. Thus, the user of this ADT will be able to create and manipulate integer stacks using code like `IntStack.pop(IntStack.push(IntStack.new(), 3))`, which returns 3, but will not be able to depend in any way on which representation it uses.

Examples of programming languages which take this approach are John Mitchell and Gordon Plotkin's SOL [11] and Luca Cardelli's Quest [1]. This approach provides data abstraction at the module level, as we have just seen. It also allows *separate compilation*, the ability to type check and compile individual module implementations using only the interfaces of the modules they reference, because modules depend only on interfaces, not on implementations. This approach is also compatible with modules as first-class values.

Unfortunately, because this approach obscures the identity of *all* type components, it precludes most interesting uses of higher-order features. As an example, first imagine creating an abstraction of an "ordered type" for use as a parameter to ordered dictionaries and the like:

```
ORDERED = interface
              type T;
              val  cmp: T*T -> int;
          end;
```

(I.e., An ordered type is just a type `T` combined with an operation `cmp` that returns `-1`, `0`, or `+1` depending on the ordering relationship between two objects of type `T`.)

Second, because we often want to extend an ordering on a type `T` to an ordering on the type list of `T` (using lexical ordering), imagine writing down the code to do this in general as a functor:

```
LexOrd = functor (o:ORDERED):ORDERED
             module
                 type T   = o.T list;
                 val  cmp = ... o.cmp ...
             end;
```

Third, imagine trying to test the new code by trying it out on the built-in integer type:

```
IntOrd = module type T = int; ... end;

IntListOrd = LexOrd(IntOrd);
```

The code runs fine, and we end up with a new ordered type, `IntListOrd`. Unfortunately, the new type is useless because it has no relation to the built-in integer type: `IntListOrd.T` is an abstract type, unrelated to `int list`. This means that we cannot create any values of this type.

The problem with this example is an instance of a more general problem: there is no way in the opaque approach to build modules "containing" other types; we can only build modules containing unrelated new types (e.g., `IntOrd.T` $\neq$ `int`). This fact means that we cannot have type abbreviations and that we cannot extend ADTs by adding a few new operations to an existing ADT. Another consequence is that many of the more useful idioms using higher-order features are inexpressible [9].

3.2 The Transparent Approach

Unlike in the opaque approach, under the transparent approach, module and functor boundaries are transparent, allowing all information about the identities of type components to pass through them; the transparent approach does this by inspecting module implementations to determine the actual identity of type components. Because the transparent approach requires access to the implementations of all the modules a module refers to in order to compile that module, it cannot provide separate compilation.

The transparent approach also cannot provide data abstraction at the module level. For example, under the transparent approach, in the previous `IntStack` example the type `IntStack.T` is not abstract: we know that `IntStack.T = int list ref` and can manipulate integer stacks using reference and list primitives. Also, the transparent approach is incompatible with treating module as first-values because there is no way to determine the actual identity of types when they depend on information available only at runtime. I show in my dissertation that unsoundness results from attempting this.

However, the transparent approach does allow modules to "contain" other types. For example, the `LexOrd` example works fine under the transparent approach with `IntOrd.T = int` and `IntListOrd.T = int list`. This fact allows most of the interesting use of higher-order features to be used under the transparent approach. Examples of programming languages that take this approach are David MacQueen's DL [9]; Robert Harper and John Mitchell's XML [10]; and Robert Harper, John C. Mitchell, and Eugenio Moggi's λ^{ML} [5]

4 My Approach

Under my approach, which I call the *translucent* approach, module and functor boundaries are "translucent," being opaque in some places and transparent elsewhere. Where the boundaries are opaque is controlled by the programmer using interfaces. Each boundary has an associated interface, which specifies which type component's identity information can be seen through the boundary.

In order to allow this specification, interfaces in my approach are extended to allow specifying the identity of type components. For example, the following interface for a hash table ADT specifies the type identity of the element type (`elem`) but not that of hash tables (`T`):

```
HashTable : interface type elem = int;
                      type T;
                      ...
            end
```

Using this ability, the programmer specifies boundary properties by giving, in the interface associated with a boundary, the type identities of only those components she wishes the boundary to be transparent to; all other type components will be treated opaquely. Thus, in the `HashTable` example above, the boundary will be transparent for the `elem` component and opaque for the `T` component.

Because the programmer is required to provide the actual type identity of transparently-treated type components in the interface (the type checker checks that the type identities provided in the interface are correct when the interface is assigned to a module), the type checker does not need to inspect the implementation of modules to determine the identity of their type components; it can just believe the module's interface. This fact means that modules depend on only the interfaces of other modules; my approach is thus able to support separate compilation without any trouble.

If the programmer does not supply an interface for a *simple* module (**module** ... **end**), then the type checker infers a fully transparent interface by inspecting the module's implementation. For example, under my approach, `IntOrd` would have been assigned the following interface:

```
interface type T = int; val cmp: T*T -> int end
```

Using a shorthand notation due to Xavier Leroy [7], this interface could also be expressed as `ORDERED` **with** `T=int`. (Leroy's with notation is a syntactic shorthand that expresses the result of adding information about type identities to an existing interface.)

Thus, under the transparent approach, the default is full transparency. Subtyping in my system permits forgetting type identities; for example, `ORDERED` **with** `T=int` is a subtype of `ORDERED`. This fact means that the programmer can use type coercions (M : **interface** ... **end**), to increase opaqueness where and when desired. For example, the following code creates an initially transparent `HashTable` module then makes opaque just the `T` type component:

```
HashTable = (module type elem = int;
                    type T = (string*elem) array;
                    ...
             end
            : interface type elem = int;
                        type T;
                        ...
            end);
```

This technique provides module-level data abstraction under my approach.

The situation for non-simple module expressions without interfaces is more complicated, particularly because modules under my approach are treated as first-class values. Briefly, by the use of some clever type rules, an interface that is as transparent as possible given soundness and reasonable compile-time information[2] constraints is inferred. Code that does not involve using modules as first-class values, *higher-order functors* (functors that take or return other functors as arguments), or type coercions will be given a fully transparent interface. This category includes all programs written in non-extended SML.

Code using modules as first-class values may automatically lose type identity information in order to avoid potential unsoundness. Consider again, for example, the code to select a dictionary at runtime based on the number of items needed:

```
Dictionary = if n<20 then LinkedList else HashTable;
```

Suppose the programmer gave fully transparent interfaces to `LinkedList`, say `DICTIONARY` **with** `T=... list`, and `HashTable`, say `DICTIONARY` **with** `T=... array`. Because the type checker does not know what `n`'s value will be at runtime, it cannot safely determine `Dictionary.T`'s type identity. Accordingly, it must be conservative and assign `Dictionary` the more opaque interface `DICTIONARY`, which makes `Dictionary.T` abstract.

No special type rules are needed to implement this behavior: Because the normal **if-then-else** type rule requires the **then** and **else** branches to have the same type and `LinkedList` and `HashTable` have different types, the type checker is forced to use subsumption to coerce both branches to a common supertype; all such types make `Dictionary.T` opaque. It is because of this ability to make types abstract when they cannot be determined at compile-time that my approach is able to treat modules as first-class values without risking unsoundness.

Standard ML has a useful feature called *type sharing* that is used in interfaces to require that (sub)-component types are equal. For example, the following interface specifies that any module with it as an interface has equal `T`, `U`, and `V` types:

```
interface type T;
          type U;
          type V
          sharing type T = U = V
end
```

The same effect can be gotten under my approach by using a series of **with**'s:

```
interface type T;
          type U;
          type V;
end with U=T with V=T
```

This basic idea — picking the type in an specified equivalence class with maximal scope and then setting the other types in that class equal to it using **with** statements — can used to translate any type sharing specification from SML into my system.

[2]In particular, the type checker makes no attempt to determine the run-time branching of conditionals, instead assuming that they could branch either way at any time.

Summarizing, under my approach the identity of type components start out fully visible; this visibility can later be obscured either automatically when modules are used in a first-class manner in order to ensure soundness, or manually when the programmer inserts a type coercion in order to produce abstraction. This controlled visibility permits data abstraction where desired and modules as first-class values without limiting the uses of higher-order features that depend on modules "containing" other types. My approach also supports separate compilation because modules depend only on interfaces.

Since my approach gives the programmer control over the degree of visibility, she can choose opaqueness or transparency as needed, getting the best of both worlds. Indeed, because translucency permits the programmer to mix opaqueness and transparency in the same module (e.g., the interface for `HashTable`), some new uses of higher-order features become possible; see Chapter 3 of my thesis for one example.

5 Translucent Sums

My approach to building higher-order module systems is based on type theory and the λ-calculus. In it modules are a particular kind of value and interfaces are a particular kind of type. These choices yield a simple and uniform design.

My recipe for constructing a higher-order module system is as follows: start with Girard's F_ω [3], a powerful type theory, to model the core language (the part without any module features); add *translucent sums* to model modules; add dependent functions to model functors (this choice allows a functor's result type to depend on its input argument); and, finally, add a notion of subtyping to model module implementation-interface matching.

The keystone here is the new type-theoretical construct I call a *translucent sum*; the other constructs used in the recipe are from the extensive type-theory literature. A translucent sum is an existential type (also called a weak sum) that has been enriched in several ways. First, its types have been extended to allow optionally specifying information about the identity of its type components; the equality relation on types is extended to use this information. Second, it has been generalized to allow any number of components, to allow named components, and to allow dependencies on previous components. And, finally, third, the subtyping on its types has been extended to allow forgetting information about type-component identities, reordering components, and dropping components. (The later two subtyping abilities allow a module with extra or differently ordered components to still be used as a functor argument.)

Like existentials, translucent sums are ordinary values. Because translucent sums are used to model modules, this fact has a number of important consequences: modules are first-class values, modules can be components of other modules (a module is just another value so no special handling is required), functors are simply ordinary functions and thus first-class values as well, and higher-order functors exist (F_ω has higher-order functions). If the ability to have modules as first-class values was not needed or desired, then the system could be stratified by adding a separate level for constructs dealing with modules; translucent sums could then be restricted to this level. This change would make modules second-class values, but would also greatly simplify many of the associated proofs of the system's properties.

Because some components of translucent sums are types, some translucent sum expressions have to be allowed to occur in types so that these type components can be used; this fact means that types can depend on term variables in my system. An important question here is exactly what sorts of translucent sum expressions to allow in types. The answer impacts the ease of type checking strongly: Type checking requires being able to decide questions like does M`.T = `N`.T` where M and N are translucent sum expressions that are allowed to occur in types. Reasonable type checking requires that deciding these questions not require having to evaluate general program expressions. (If the type checker had to evaluate general code, it would take too long to be useful in many cases.)

My answer is to allow only translucent-sum *values*, not arbitrary computations in types; for this purpose, I consider values to be given by the normal grammar for call-by-value values extended with rules for term variables and component selections on values (M`.x`). (For example, **module val** `x=M.x` **end** is a value.) I show that by repeatedly reducing selections (e.g., replacing **module type** `T=int` **end**`.T` with `int`) in types, well-typed values in types can always be reduced to (repeated) selections on term variables (e.g., `x.y.z`). Type equality is easy for this case since equality on type components of such values is syntactic equality extended with any type abbreviations supplied by the term variable's interface; no evaluation of arbitrary code is required at any time.

One important consequence of this answer is that substitution of general terms is not possible: Because types may contain term variables, substitution of M for a term variable in a term can also involve substituting M into a type; this limits the set of terms that can be substituted to the set that may appear in types. This fact means that my approach is not compatible with call-by-name evaluation.

Allowing arbitrary terms in types in my system is not just problematic because of the resulting difficulty of deciding type equalities; this choice also leads to unsoundness. To see this fact, first consider the following definitions:

```
Int =  module type T = int;
                val  v = 3;
                val  f = negate;
       end;

Bool = module type T = bool;
                val  v = true;
                val  f = not;
       end;
```

Each of these modules packages up a type, `T`, a value of that type, `v`, and a function on values of that type, `f`.

Second, consider the following piece of code that I shall abbreviate by Δ. Note: this abbreviation is not a definition within the language; the code on the right is to be mentally substituted wherever Δ is used from now on.

$\Delta \equiv$ **if** `rnd()%2 = 0` **then** `Int` **else** `Bool`

The `rnd` function here is a pseudo-random number generator that returns an integer; accordingly, at runtime Δ`.T` is equal to the type `int` half the time and the type `bool` the rest of the time.

Now, if general terms could appear in types, the following would type check:

```
Δ : interface type T = Δ.T;
               val  v : T;
               val  f : T->T;
             end
```

This typing would allow the following code to type check:

```
X = Δ;
Y = Δ;

X.f(Y.v)
```

But, when run, half the time this code evaluates either `not(3)` or `negate(true)`, which generate runtime type errors!

6 The Kernel System

The kernel system forms the core of my dissertation. It provides a concrete example of how my approach can be used to build a higher-order module system. Type theory is something of a black art; how exactly to arrange the system and its associated proofs so they are tractable is one of the major contributions of my dissertation. Many of the problems encountered in proving the system's properties are new, lacking any existing solution in the literature. The proofs are especially difficult because the kernel system contains a number of cyclic dependencies between its parts. For example, the type validity judgment depends on the type equality judgment, which in turn depends on the type validity judgment.

As I mentioned earlier, the kernel system has been simplified so as to make its proofs as easy as possible without losing any results. Among other simplifications, translucent sum expressions in kernel-system types are limited to (repeated) selections on term variables[3] and translucent sums have been factored into two simpler constructs, a standard dependent sum and a new construct called a *reified constructor* (see Chapter 8 of my dissertation for details). The more complicated version I described in the previous section can be recovered from the kernel system by adding a pre-processing elaboration stage. I have also chosen to move the problem of reordering and dropping translucent sum components to this elaboration stage. I did this both to simplify the system and because there is some uncertainly about what the best choice of subtyping rules is for handling component reordering.

In addition to giving the syntax, the typing rules, and the semantics for the kernel system, I also give formal proofs of all the important properties for a programming language. In

[3]Values can still be substituted, however, because the kernel-system substitution operator does selection reductions as it substitutes terms.

particular, I prove that type checking modulo subtyping is decidable and that the system is sound even in the presence of side effects. Unfortunately, subtyping, and hence full type checking, is only semi-decidable; this is unlikely to cause problems in practice though because the semi-decidability is in the right direction (only ill-typed programs can cause looping), the smallest known looping example requires more complexity than any normal human-written program is likely to have, and a simple time limit can be used to detect bad programs (programs either ill-typed or so complicated that they cannot be typed checked in reasonable time).

7 Related Work

Along with the work I have already mentioned in Section 3, the most directly relevant work to mine is Xavier Leroy's work on *manifest types* [7]. This work, done independently, uses similar ideas but differs most fundamentally from mine in that his system treats modules as second-class values. This choice greatly simplifies the theoretical complexity of his system and holds out the possibility of a decidable type system if named interfaces are prohibited. His system also differs from mine in that his system is based on Damas-Milner style polymorphism [2] and is implicitly typed, while my system is based on Girard's F_ω and is explicitly typed. He does not provide either a proof of soundness or a provably-correct type-checking procedure[4] for his system.

8 Contributions

This dissertation is a major step forward in the design of higher-order module systems. For the first time it is now possible to have a higher-order module system combining the best of the previous approaches with none of their disadvantages, resulting in a system with all of the following features:

- Data abstraction at the module level
- Support for Separate compilation
- Modules as first-class values
- Modules can "contain" other types
- Type abbreviations
- Transparency and opaqueness can be mixed in a single module
- Higher-order functors
- Type sharing

[4]He did give a type checking procedure and a "proof" of its correctness in his paper, but both were later discovered to be flawed.

I have developed the necessary machinery and established the validity of this approach by showing

- How to arrange the system and its associated proofs so they are tractable
- The system's soundness even in the presence of side effects
- How to do type checking effectively

The negative results about soundness under certain extensions and the decidability of subtyping (and hence type checking) are also contributions of my dissertation.

References

[1] Luca Cardelli. Typeful programming. Technical Report 45, DEC Systems Research Center, 1989.

[2] Luis Damas and Robin Milner. Principal type schemes for functional programs. In *Ninth ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[3] Jean-Yves Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieure*. PhD thesis, Université Paris VII, 1972.

[4] Robert Harper, Robin Milner, and Mads Tofte. The definition of Standard ML (version 3). Technical Report ECS–LFCS–89–81, Laboratory for the Foundations of Computer Science, Edinburgh University, May 1989.

[5] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM Symposium on Principles of Programming Languages*, San Francisco, CA, January 1990.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-hall, Inc., 1978.

[7] Xavier Leroy. Manifest types, modules, and separate compilation. In *Proceedings of the Twenty-first Annual ACM Symposium on Principles of Programming Languages, Portland*. ACM, January 1994.

[8] Barbara Liskov, Russell Atkinson, et al. *CLU Reference Manual*, volume 114 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[9] David MacQueen. Using dependent types to express modular structure. In *Thirteenth ACM Symposium on Principles of Programming Languages*, 1986.

[10] John Mitchell and Robert Harper. The essence of ML. In *Fifteenth ACM Symposium on Principles of Programming Languages*, San Diego, California, January 1988.

[11] John C. Mitchell and Gordon Plotkin. Abstract types have existential type. In *Twelfth ACM Symposium on Principles of Programming Languages*, 1985.

[12] I. C. Pyle. *The Ada Programming Language.* Prentice-Hall International, 1981.

[13] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1987.

[14] Niklaus Wirth. *Programming in Modula-2.* Texts and Monographs in Computer Science. Springer-Verlag, 1983.