Towards a Notion of Module for Data Abstraction

Dennis Volpano

TR 89-991
May 1989

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

The author acknowledges support from the NSF under grant MCS-83-03927, and joint support from the NSF and DARPA under grant ASC-88-00465.

Towards a Notion of Module for Data Abstraction

Dennis Volpano

Department of Computer Science
Upson Hall
Cornell University
Ithaca, NY 14853-7501

5 May 1989

Abstract

Traditionally, programming languages support data abstraction through some kind of module construct for partitioning large systems into manageable units. These constructs typically control access to data since program decomposition is usually guided by information hiding. As mechanisms for encapsulating implementations of data types, however, such constructs are too inflexible. Substituting one implementation (module) for another, in a client, may require the client to be revised for reasons related to representation. A more flexible notion of module is presented that is designed solely for the purpose of encapsulating implementations of data types.

D.3 [**Software**]: Programming Languages; D.3.2 [**Programming Languages**]: Language Classifications - *applicative languages*; D.3.3 [**Programming Languages**]: Language Constructs - *abstract data types, modules, packages*; D.3.4 [**Programming Languages**]: Processors - *compilers*; F.3 [**Theory of Computation**]: Logics and Meanings of Programs; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs - *type structure*

Additional Key Words and Phrases: representation independence, modules, functional programming, types

---

The author acknowledges support from the NSF under grant MCS-83-03927, and joint support from the NSF and DARPA under grant ASC-88-00465.

1. Introduction

Many modern programming languages (Ada, Modula-3, C++) offer module constructs for partitioning large systems into manageable units. Decomposition is usually guided by the principle of information hiding, so it is not surprising that these constructs, known as *packages* (Ada) [2], *modules* (Modula-3) [8], and *classes* (C++) [26], can also be used to control access to data. But, they are too inflexible as mechanisms for supporting data abstraction.

A traditional module is divided into two parts: an *implementation*, where operations are implemented in terms of a private representation, and an *interface*, or module signature, that reveals just enough information about the implementation to enable programmers to use it. All access to the implementation is through the interface, so any types or operators defined in the module are hidden from clients of the module unless declarations for them appear in the interface. For example, using a typical syntax, we might define a module *M* to encapsulate an implemention of a set of integers:

module *M* is
interface
    type *intset*
    function *excl* (*s* : *intset* , *e* : *int*): *intset*
    function *incl* (*s* : *intset* , *e* : *int*): *intset*
implementation
    (*excl s e*) *and* (*incl s e*) *implemented*

As specified by the interface, or signature, of *M*, *M* encapsulates a representation called *intset* and implementations of expressions (*excl s e*) and (*incl s e*) in terms of *intset*. The former returns a set that is *s* with element *e* excluded; the latter is a set that is *s* with *e* included. Operations *excl* and *incl* must be interpreted as functions in any program that uses module *M*. For example, here is a client of *M*:

(1.1) **var** *u*: *M.intset*
(*M.incl* (*M.excl u m*) *n*)

Suppose we have another module *N* that also encapsulates an implementation of a set of integers but differs from *M* in that it implements statement *s* := (*excl s e*) instead of expression (*excl s e*). It may be possible to implement the statement, which usually takes the form of a procedure, more efficiently given the private representation used in *N*. For example, *M* might use a 32-bit word to represent a set of integers while *N* might use a heap. The interface of *N* specifies that *N* implements statement *s* := (*excl s e*), rather than expression (*excl s e*), so the client

(1.2) **var** *u*: *N.intset*
(*N.incl* (*N.excl u m*) *n*)

obtained from (1.1) by substituting *N* for *M*, is not well-formed. The client must be revised to accomodate the interface of *N*. Some might argue that such a revision is reasonable because a new interface is, in essence, the definition of a new abstract data type in this paradigm. However, if *intset* is viewed as an abstract data type with operations *excl* and *incl* as functions, in the mathematical sense, then client (1.2) merely reflects a change in the representation of *intset* in client (1.1). From this point of view, it is unreasonable to expect the client to be revised because it would be due to a change in the representation of an abstract data type.

The traditional notion of module makes no distinction between a module and the abstract data type it implements. Programs are expressed in terms of module interfaces. This leads to a programming paradigm that prevents a program from being developed and proved correct at an abstract level, i.e. independently of implementations [10,11]. Also, modules implement the same abstract data type as long as their interfaces remain the same. This is the extent to which one can view a module as one of many implementations of a single abstract data type. But interfaces can change for subtle reasons related to

representation, such as a change in the calling mode of a procedure parameter. As a consequence, traditional modules tend to be inflexible in that they cannot be interchanged freely within a client.

There is a need for a language with more flexibility in describing and using implementations of abstract data types. This paper describes such a language, called TL. TL is a typed, functional language in which a program may be expressed with functions whose implementations are separately specified in modules. A module need not implement functions only, however. It may implement an assignment statement in which the result of a function application is assigned to a variable, but the decision to do so is private in that it has no effect on whether a client of the module is well-formed. For example, if M and N were expressed as TL modules then we would define a TL function

$$\lambda u : set\ int.\, \lambda m, n : int.\ incl\ (excl\ u\ m)\ n$$

and specify that the representation of (*set int*) is to come from module M by annotating the type of u with $/M$, as in $u : (set\ int)/M$. Annotating the type with $/N$, instead, indicates that the representation should come from module N. The function body remains the same, only a module annotation changes. In effect, N becomes more flexible because it can be substituted for M without causing the function, or client, to become ill-formed. This is a major step towards achieving representation independence, the primary goal of data abstraction.

The design of TL has been influenced by other languages, most notably, Polya, under development at Cornell, Standard ML with modules [15], and SOL [20], an extension of Reynold's polymorphic lambda calculus [23,24] with existential types. A TL program is an expression abstracted with respect to representations of the types over which it is defined. This is what makes a TL program abstract. A concrete instance of a TL program is obtained by applying it to one or more representations. Thus, through lambda abstraction, as in the lambda calculus, we achieve representation independence, and through its complement, application, we arrive at concrete instances for particular representations. Although the basic idea of achieving data abstraction in this way has long been recognized [23], its development as described in this paper is new.

A representation of a type is packaged in a module together with implementations of various operators on the type that use the representation. From modules, an implementation of TL compiles concrete instances of TL programs, that is to say, programs committed to particular representations of their data types. A TL program that is not committed to representations of all of its data types is a program for which some technique that automatically selects representations could be used. Automatic selection of data representations has been explored in other languages such as SETL [25], but it is not attempted in TL. Committing to representations, however, does not guarantee successful compilation. Some operators may not be implemented for the chosen representations. In practice, an operator may be implemented for some representations but for others, it may be too inefficient to implement.

We begin with an overview of TL in section 2, discussing properties of its type system and value space. Section 2.2 describes representation independence in TL as lambda abstraction on expressions with respect to modules. Modules and their properties are described in section 3, and a prototype compiler for TL is discussed in section 4. Section 5 concludes with some remarks about the relationship of TL with other programming languages.

2. The typed language TL

TL is a typed, functional language. Every expression has a type that contains all values produced by evaluating it. The *type formulas* of TL are given by the following abstract grammar.

$$\tau ::= \sigma \mid \mathbf{Type}_1 \mid c \sim \mid \Pi t : \tau.\, \tau'$$
$$\sigma ::= t \mid (c\ \sigma_1, \ldots, \sigma_n)\,[/M]$$

where c may be any type constructor, t any type variable, and M any module name. Brackets signify that

$/M$ is optional. For a type t and module M, t/M is called a *representation type*. It is the type that represents t in M. More will be said about the meaning of such types in section 2.2. Modules that represent type constructor c have type $c\ \sim$, which conveys a binding of c to a representation. $\Pi t{:}\tau.\tau'$ is the dependent function type. It is the type of all functions that map a value x of type τ into a value of type τ' where τ' may depend on x. If τ' does not depend on values of type τ then $\Pi t{:}\tau.\tau'$ degenerates to type $\tau \to \tau'$. See [16,17,21] for presentations of type theory and [4,5,7,14,22] for other descriptions of the role of dependent types in programming languages.

The TL type universe is stratified into two levels: $\mathbf{Type}_1$ and $\mathbf{Type}_2$. The closed type formulas derivable from σ denote types that belong to $\mathbf{Type}_1$, the universe of *first-order* types. Members of types in $\mathbf{Type}_1$ are first-class values. $\mathbf{Type}_1$ is itself a type, the type of first-order types, but it does not belong to itself. It belongs to $\mathbf{Type}_2$ and, consequently, first-order types are not first-class values. The type system is therefore irreflexive [5,6,18].

The type formulas derivable from τ denote types that belong to $\mathbf{Type}_2$, the universe of *second-order* types. $\mathbf{Type}_2$ contains first-order types, the type of first-order types, module types, and types formed from them by closing under dependent-type former Π. Members of types in $\mathbf{Type}_2$ are second-class values, and include first-class values, first-order types, modules, and functions.

Unlike the type systems of SOL, Russell [12], and Pebble [4], TL's type system is *predicative*. In a predicative type system, members of types in a universe at level n, say U_n, are level $n-1$ objects, so only values of types in U_1 are first-class values. An *impredicative* type system, on the other hand, usually yields a larger set of first-class values because types at levels U_1, U_2, and so on, in a predicative system, are contained in a single, unstratified, universe at level 1.

2.1. Functions and application

With the exception of the conditional expression, TL has no built-in functions. Primitive functions are declared in TL merely by specifying their types. These functions are candidates for implementation within modules. For example, if *set* is a unary type constructor, then the following functions are primitive polymorphic operators on *set*.

$$\begin{aligned}
&\mathit{emptyset} : \Pi t{:}\mathbf{Type}_1.\ \mathit{set}\ t\\
&\mathit{size} : \Pi t{:}\mathbf{Type}_1.\ \mathit{set}\ t \to \mathit{int}\\
&\mathit{min} : \Pi t{:}\mathbf{Type}_1.\ \mathit{set}\ t \to t\\
&\mathit{incl}, \mathit{excl} : \Pi t{:}\mathbf{Type}_1.\ \mathit{set}\ t \to (t \to \mathit{set}\ t)
\end{aligned}$$

In TL, we adopt explicit-type function declarations, but then allow implicit-type function applications where type arguments are omitted by the user and recovered through inference.

All other functions are defined using lambda abstraction. The type deduction rules for lambda abstraction and application are given below. If it can be deduced from an environment E that some expression e has type α, then we write this in our deduction rules as $E \vdash e{:}\alpha$. An environment is a set of assumptions that associates types with identifiers. We use the notation $E, x{:}\alpha$ to denote environment E extended by the assumption that x has type α, and $s\ \{u := v\}$ to denote the result of substituting v for all free occurrences of u in term s.

$$\frac{E \vdash \tau{:}\mathbf{Type}_2\ \ E, x{:}\tau \vdash b{:}\tau'}{E \vdash (\lambda x{:}\tau.\, b){:}\,\Pi x{:}\tau.\tau'}$$

$$\frac{E \vdash e{:}\,\Pi x{:}\tau.\tau'\ \ E \vdash e'{:}\tau}{E \vdash (e\ e'){:}\,\tau'\{x := e'\}}$$

Applications of the form $(\lambda x.\ b)\ e$ can be expressed in TL as (**let** $x = e$ **in** b). The operational meaning of **let** follows directly from function application. The type deduction rule for **let** is

$$\frac{E \vdash e:\tau \quad E, x:\tau \vdash e':\tau'}{E \vdash (\textbf{let}\ x = e\ \textbf{in}\ e'):\tau'}$$

Expression (**letrec** $x = e$ **in** b) is an abbreviation for (**let** $x = \textbf{rec}(x)\ e$ **in** b), where **rec** obeys the reduction rule $\textbf{rec}(x)\ e \Leftrightarrow e\ \{x := \textbf{rec}(x)\ e\}$. The type deduction rule for **rec** is

$$\frac{E, x:\tau \vdash e:\tau}{E \vdash (\textbf{rec}(x)\ e):\tau}$$

2.2. Representation independence

Representation independence is achieved in TL through abstracting expressions on module variables. An expression e of first-order type t is lifted to an abstract expression by the function

$$\lambda M:(t\ \sim).\ e:t\ /M$$

For instance, constant zero of type *int* becomes abstract by the function

$$\lambda M:(int\ \sim).\ 0:int/M$$

which has type $\Pi M:(int\ \sim).\ int/M$. It is a function that when applied to a module *INT* of type *int* ~, i.e. a module that represents *int*, produces a constant of type *int*/*INT*. As another example of representation independence, consider a function *Replace* defined by

$$Replace = \lambda M:(set\ int\ \sim).\ \lambda u:(set\ int)/M.\ \lambda m, n:int.\ incl\ (excl\ u\ m)\ n$$

Suppose *BITV* is a module of type *set int* ~ that represents a set by a bit vector. Then (*Replace BITV*) is a concrete instance of *Replace* whose domain is (*set int*)/*BITV*. Another concrete instance is given by (*Replace* (*HEAP* 32)) where module *HEAP* offers a heap representation of sets that is parameterized on heap size. Type *int* is an implicit argument in the application of *excl* and in the application of *incl*. Module *M* is also an implicit argument in each of these applications.

A primitive function in TL is regarded as an expression implicitly abstracted on a module for each data type it manipulates. For example, *incl* and *excl* are each regarded as a primitive function abstracted on a module of type *set* ~:

$$incl, excl : \Pi M:(set\ \sim).\ \Pi t:\mathbf{Type}_1.\ (set\ t)/M \rightarrow (t \rightarrow (set\ t)/M)$$

The definition of *Replace* is equivalent to

$$\lambda M:(set\ int\ \sim).\ \lambda u:(set\ int)/M.\ \lambda m, n:int.\ incl\ M\ (excl\ M\ u\ m)\ n$$

Module parameters are implicit in the declarations of primitive functions and explicit in the declarations of lambda abstractions. But module arguments may be omitted in applications of either kind of function. Like missing type arguments, TL relies on inference to recover missing module arguments.

It is quite common for a concrete instance of a TL function to be formed by applying the function to more than one module as the following example illustrates. Let *tree* be a unary type constructor, and *val* and *node* be primitive functions with types

$$val : \Pi\, t{:}\,\mathbf{Type}_1.\ tree\ t \rightarrow t$$
$$node : \Pi\, t{:}\,\mathbf{Type}_1.\ tree\ t \rightarrow (t \rightarrow (tree\ t \rightarrow tree\ t))$$

(*val n*) is the value at node *n*, and (*node l v r*) is the binary tree with root value *v* and left and right subtrees *l* and *r*. We define a function *Huffman* that constructs a binary tree with minimum weighted external path length [13]:

```
Huffman = λM,N. rec(H) λs: (set (tree int)/N)/M.
        if (size s) = 1 then s
        else let l = min s in
                let r = min (excl s l) in
                        H (incl (excl (excl s l) r)) (node l (val l + val r) r)
```

From a set of binary trees, with weights stored at the roots, *Huffman* produces a set containing a single binary tree with the desired property. At each step, the two trees with smallest root weights are replaced by a new tree with these two trees as left and right subtrees, and with a root that stores the sum of the two weights. If *TREE* is a module of type *tree* ~, then

Huffman (*HEAP* 100) *TREE*

is a concrete instance of *Huffman*. As we shall see, the use of *HEAP* to represent a set is a wise choice because it offers a constant-time implementation of *min* and it affords control over the size of a heap.

In a TL module, a first-order type *t* is represented by another first-order type *r*. If *M* is such a module then *t*/*M* is the type of values in *r* that satisfy the *coupling invariant* associated with *M*. The coupling invariant is a predicate that establishes a relationship between *t* values and *r* values, and indicates which *r* values may be interpreted as representations of *t* values [10]. Examples of coupling invariants are given in the next section.

Although representation types are types, a strong separation is maintained between them and the types they represent. Representation types have no role in type checking. The application of a function whose domain is *t*/*M* to a value with type *t*/*N* is well-typed. Compatibility among representation types is not a well-formedness issue.

3. Modules

A module in TL is not a program-structuring construct. Unlike modules in other languages, such as Modula-3, it has no executable body. It is only a means to package a representation of a type together with implementations of operators over the type expressed in terms of the representation. Representations are data structures typically with operations that update the structures. Such updates are easily expressed in a language with assignment statements. Therefore, modules are expressed in a typical imperative language whose data types belong to $\mathbf{Type}_1$. This approach allows us to exploit, in a functional language, the efficiency gained through performing incremental updates at the representation level.

A TL module has the form

module *id* = *type–binding* { | *imp–binding* } **end** *id*

A module contains a type binding and zero or more implementation bindings. A type binding has the

form t ~ r where t is a type constructor and r is a type that represents t. A type binding supplies the types that may be used in the implementation bindings that follow it. A module with type binding t ~ r has type t ~. An implementation binding has the form p ~ q where p is an expression containing a single primitive function, or a statement that assigns such an expression to a variable, and q is, respectively, an expression or statement that implements p. For example, with declarations of primitive functions *pr*, *fst*, and *snd*,

$$pr : \Pi\, a,b : \mathbf{Type}_1.\ a \rightarrow (b \rightarrow pair\ a\ b)$$
$$fst : \Pi\, a,b : \mathbf{Type}_1.\ pair\ a\ b \rightarrow a$$
$$snd : \Pi\, a,b : \mathbf{Type}_1.\ pair\ a\ b \rightarrow b$$

we might define a module *PAIR* with three implementation bindings:

```
module PAIR =
pair 'a 'b      ~   R = record m:'a; n:'b end
| pr x y        ~   R [x,y]
| fst z         ~   z.m
| snd z         ~   z.n
end PAIR
```

whose coupling invariant is

$$(x,y) : pair = z : pair\ /PAIR \ \equiv\ z.m = x \wedge z.n = y$$

Each occurrence of z in expressions $(fst\ z)$ and $(snd\ z)$ has type *pair*, which is represented by a record named R within the module. Therefore, z denotes a record in the implementations of these expressions. Type synonym R is used in the record construction $R\ [x,y]$ to represent *pr* and illustrates a dependency between two bindings. This construction is a record with field m bound to x and field n bound to y.

A TL module may implement an assignment of an expression to a variable. For example, here is a module that implements the assignment of expression $(pr\ x\ y)$ to a variable:

```
module MYPAIR =
pair 'a 'b      ~   record m:'a; n:'b end
| z :=(pr x y)  ~   z.m :=x; z.n :=y
| fst z         ~   z.m
| snd z         ~   z.n
end MYPAIR
```

The fact that *MYPAIR* implements an assignment statement does not, alone, limit the contexts in which *PAIR* can be replaced by *MYPAIR*, which makes the decision to implement the assignment a private one.

The ability to implement an assignment is critical to expressing efficient implementations because, in practice, some representations are too costly to copy and must be updated incrementally instead. For example, if *set* is represented by a heap, it is more efficient simply to update the heap when an element is included by *incl* or excluded by *excl*. The following module *HEAP* makes use of this fact by implementing assignment statements involving these two functions. *HEAP* represents a set by a heap stored in an array a in the usual way and a variable n that is the size of the set. Its coupling invariant is

$$s : set = (a,n) : set\ /(HEAP\ k) \ \equiv\ \{\, 0 \le n \le k \wedge perm\,(s, a[1\,..\,n]) \wedge heap\,(a[1\,..\,n])\,\}$$

where $perm\,(s, a[1\,..\,n])$ is true if $a[1\,..\,n]$ is precisely the elements of set s, and $heap\,(a[1\,..\,n])$ is true if $a[1\,..\,n]$ has the heap property with the minimum element stored in $a[1]$.

```
module HEAP (k:int) =
set 't              ~  record a:array 1..k of 't; n:int end
| min s             ~  s.a[1]
| size s            ~  s.n
| s := emptyset     ~  s.n := 0
| s := (incl s e)   ~  s.n := s.n + 1;
                       s.a[s.n] := e;
                       (bubble-up s.a s.n)
| s := (excl s e)   ~  var i:int := 1;
                       while s.a[i] ≠ e do i := i + 1
                       s.a[i] := s.a[s.n];
                       s.n := s.n - 1;
                       (bubble-down s.a i s.n)
end HEAP
```

HEAP is parametric in the size of the array used to contain the heap; (*bubble–up a n*) inserts $a[n]$ into the heap $a[1..n-1]$ and (*bubble–down a i n*) builds a heap from node $a[i]$ and the left and right heaps (subtrees) rooted at $a[i]$.

3.1. Typings within modules

In a TL module, every implementation binding $p \sim q$ has an associated *typing* of the form $(x_1{:}A_1, \ldots, x_n{:}A_n)\,A$ where x_i is a variable from p. A is the type of p and q, and is omitted if p is an assignment statement. With a typing, an implementation binding has the form

$$p\ (x_1{:}A_1, \ldots, x_n{:}A_n)\,A \sim q$$

Types (the A's) are expressed with representation types. For example, here is an alternative definition of module *PAIR* with explicit typings:

```
module PAIR =
pair 'a 'b                              ~  R = record m:'a; n:'b end
| pr x y
      (x:'a; y:'b) (pair 'a 'b)/PAIR    ~  R [x, y]
| fst z
      (z:(pair 'a 'b)/PAIR) 'a          ~  z.m
| snd z
      (z:(pair 'a 'b)/PAIR) 'b          ~  z.n
end PAIR
```

An occurrence of t/M in module M may be abbreviated as t, which may allow a typing to be eliminated altogether. If, after such an abbreviation, the typing associated with an implementation of a primitive function is the type specified for the function in its declaration, then the typing may be omitted, in which case the implementation has an implicit typing obtained from the declaration. For example, each occurrence of (*pair 'a 'b*)/*PAIR* may be abbreviated by dropping *PAIR*. Doing so with the typing associated with the implementation of *pr*, for example, leaves the typing

$$(x{:}'a;\ y{:}'b)\,(pair\ 'a\ 'b)$$

which is equivalent to $\Pi a, b.\ a \to (b \to pair\ a\ b)$, the type of *pr* by its declaration. Therefore, the specified typing may be omitted and the typing associated with the implementation of *pr* becomes

implicit. Likewise, the typings for the implementations of *fst* and *snd* may also be omitted, resulting in the original definition of *PAIR*.

Typings are used to distinguish multiple implementations of the same expression or assignment statement. For two different module names M and N, t/M is not equivalent to t/N because it is assumed that M and N have different coupling invariants. This is the basis for discriminating implementation bindings in TL. Typings that contain occurrences of t/M are distinguished from typings that contain t/N. If two typings are not the same, then neither are their associated implementation bindings.

Modules *PAIR* and *HEAP* are examples of modules that represent parametric types. Each gives implementations of primitive polymorphic functions such as *fst* and *min*. Sometimes it is practical to implement only an instantiation of a polymorphic function. For example, with a bit-vector representation of a set, certain set operations can be implemented efficiently for sets containing integers in the range $0..n$ for some n. The following module *BITV* makes use of this fact by representing a nonparametric type *set int*. Its coupling invariant is

$$s : set\ int = k : (set\ int)/BITV \equiv (i \in s \Leftrightarrow (and\ k\ (shift\ 1\ i)) = 2^i)$$

where *and* is bit-wise logical conjunction, and the result of $(shift\ x\ n)$ is x shifted to the left n positions with vacated bits set to zero.

```
module BITV =
set int                                  ~  int
| emptyset
        () set int                       ~  0
| incl s e
        (s: set int; e: int) set int     ~  (or s (shift 1 e))
| excl s e
        (s: set int; e: int) set int     ~  (and s (not (shift 1 e)))
| union u w
        (u, w: set int) set int          ~  (or u w)
| inter u w
        (u, w: set int) set int          ~  (and u w)
end BITV
```

Notice that within each typing, *set int* and *int* occur without module annotations. Every occurrence of *set int* is treated as an abbreviation of $(set\ int)/BITV$ because *set int* is represented in *BITV*. If *set int* were not represented in *BITV* then the typings would be incorrect because they are not expressed with representation types. Each occurrence of *int* is implicitly annotated with the name of a module that provides the standard representation of *int*, one that is acceptable across a wide range of applications. To implement the set operators using another representation of *int* that comes from, say module *INT*, would require new typings that ascribe type int/INT to e. In general, any implementation may be expressed in terms of representations from different modules.

The typings in *BITV* cannot be omitted because they convey a specialization of the set operators; specifically, the typings convey implementations of the set operators, but only on sets of integers. If the typings were omitted then the implementation bindings would have implicit typings obtained from the definitions of these operators, which would lead to incorrect typings because the implicit typings are too general.

With module *BITV*, the maximum size of a set is determined by the standard representation of *int*. In a typical, standard representation, this might be the number of bits in a word. The following module affords control over the maximum set size, but does so at the expense of implementations of union and intersection.

```
module BOOLARRAY(k : int) =
set int                          ~   array 1 .. k of bool
| s := emptyset
        (s : set int)            ~   for i := 1 to k do s[i] := false
| s := (incl s e)
        (s : set int; e : int)   ~   s[e] := true
| s := (excl s e)
        (s : set int; e : int)   ~   s[e] := false
end BOOLARRAY
```

BOOLARRAY has the coupling invariant

$$s : set\ int = a : (set\ int)\ /\ (BOOLARRAY\ k) \equiv (i \in s \Leftrightarrow a[i])$$

Unlike *BITV*, this module implements no expressions but, in TL, this alone does not limit the contexts in which *BITV* can be replaced by *BOOLARRAY*. The loss of union and intersection, however, will render *BOOLARRAY* unusable in programs that rely on these two functions.

4. A prototype compiler for TL

A TL compiler compiles concrete instances of TL programs into code generated from TL modules. A concrete instance is a first-order typed function, in other words, a function $\lambda x : \tau.\ e$ where τ: $\mathbf{Type}_1$. This rules out functions that, for example, take modules as input.

A prototype TL compiler was developed at the Oregon Graduate Center in 1986 [27], and modified in 1987 at the Microelectronics and Computer Technology Corporation (MCC) [28]. There are four major steps taken by this compiler. First, a function is checked for type correctness. If it is type-correct then nested lambda abstractions are lifted to the outer-most level, which yields new functions. The next step transforms each of these functions into tail-recursive form. In the final step, code is generated from modules.

Lifting lambda abstractions to the outer-most level, called *lambda lifting* [1], simplifies compilation by permitting uniform treatment of all functions. The process of lifting involves replacing references to non-local variables with references to new local variables so that all identifiers occurring in an expression are constants or local variables. For example, let $Huffman_{M,N}$ stand for $(Huffman\ M\ N)$ for some modules M and N:

$$\begin{aligned}
&Huffman_{M,N} = \lambda s : (set\ (tree\ int)/N)/M.\\
&\quad \textbf{if } (size\ s) = 1 \textbf{ then } s\\
&\quad \textbf{else let } l = min\ s \text{ in}\\
&\qquad \textbf{let } r = min\ (excl\ s\ l) \text{ in}\\
&\qquad Huffman_{M,N}\ (incl\ (excl\ (excl\ s\ l)\ r))\ (node\ l\ (val\ l + val\ r)\ r)
\end{aligned}$$

The inner-most occurrence of **let** is converted to the application of

$$\lambda r.\ Huffman_{M,N}\ (incl\ (excl\ (excl\ s\ l)\ r))\ (node\ l\ (val\ l + val\ r)\ r)$$

to $(min\ (excl\ s\ l))$. As it stands, this lambda abstraction cannot be lifted from the body because of the references to non-local variables s and l. However, it can be converted to a function in which only local variables are referenced:

$$F = \lambda s.\ \lambda l.\ \lambda r.\ Huffman_{M,N}\ (incl\ (excl\ (excl\ s\ l)\ r))\ (node\ l\ (val\ l + val\ r)\ r)$$

The inner-most occurrence of **let** may then be expressed as

$$F\ s\ l\ (min\ (excl\ s\ l))$$

Repeating this process for the remaining occurrence of **let** produces

$$Huffman_{M,N} = \lambda s : (set\ (tree\ int)/N)/M.$$
$$\textbf{if}\ (size\ s) = 1\ \textbf{then}\ s\ \ \textbf{else}\ G\ s\ (min\ s)$$

where G is defined by

$$G = \lambda s.\ \lambda l.\ F\ s\ l\ (min\ (excl\ s\ l))$$

$Huffman_{M,N}$ is now a first-order typed function free of nested occurrences of lambda, which makes it suitable for transformation into tail-recursive form.

A function is tail-recursive if no function call is preceded by a call to a non-primitive function. For example, $Huffman_{M,N}$ is tail-recursive. But, suppose *min* were replaced by a non-primitive function *Min* defined by $\lambda s.\ min\ s$. $Huffman_{M,N}$ becomes

$$Huffman_{M,N} = \lambda s : (set\ (tree\ int)/N)/M.$$
$$\textbf{if}\ (size\ s) = 1\ \textbf{then}\ s\ \ \textbf{else}\ G\ s\ (Min\ s)$$

and is no longer tail-recursive because a call to *Min* precedes a call to G. To convert such a call into the appropriate form, a continuation is introduced. The application of G is represented by a continuation, and G is replaced by a call to a new function *MinC* for which the continuation is supplied as an argument [29,30]:

$$Huffman_{M,N} = \lambda s : (set\ (tree\ int)/N)/M.$$
$$\textbf{if}\ (size\ s) = 1\ \textbf{then}\ s\ \ \textbf{else}\ MinC\ s\ (\lambda v.\ G\ s\ v)$$

MinC is a function such that for any continuation γ,

$$MinC\ s\ \gamma = \gamma\ (min\ s)$$

and $(\lambda v.\ G\ s\ v)$ is a continuation. The value of a continuation expression $\lambda v.\ e$ is a closure formed by replacing each free variable of e by the value it had at the time the expression was evaluated (known as proper binding). A continuation is represented in the generated code as a stack frame containing a return address and values bound to any free variables in the continuation. Consequently, generated code never relies on a system run-time stack.

It is important to note that this transformation is not limited to any particular recursion schemes. But, it may produce tail-recursive functions whose evaluations still require an unbounded amount of space. This may even be true for functions that can be evaluated in bounded space, so in this sense, the algorithm is not optimal.

After all functions produced by the lambda-lifting step are transformed into tail-recursive form, code is generated for them based on implementations prescribed by modules. Which implementations are selected for use in the code-generation step, depends on representation types. Every occurrence of a primitive function is an implicit reference to an implementation binding by virtue of all functions being concrete. An implementation binding prescribes an implementation of a primitive function applied to particular modules. For example, with the type of *min* as

$$min: \Pi M: (set\ \sim).\ \Pi t: \mathbf{Type}_1.\ (set\ t)/M \rightarrow t$$

the implementation binding for *min* in module *HEAP* is equivalent to

$$min\ (HEAP\ k)\ \sim \lambda t: \mathbf{Type}_1.\ \lambda s: (set\ t)/(HEAP\ k).\ s.a\ [1]$$

for some k. This binding gives an implementation of polymorphic function (*min* (*HEAP* k)), not *min*. The binding is referenced implicitly by any occurrence of *min* applied to a set of type (*set* t)/(*HEAP* k) for some type t. For example, s has type

$$(set\ (tree\ int)/TREE)/(HEAP\ k)$$

in the context of (*Huffman* (*HEAP* k) *TREE*) so the occurrence of *min* in (*min* s) is an implicit reference to this binding. In other words, (*min* s) is equivalent to

$$min\ (HEAP\ k)\ (tree\ int)/TREE\ \ s$$

Similarly, the occurrence of *excl* in (*excl* s l) is an implicit reference to the implementation binding for *excl* in *HEAP*. Unlike the previous binding, however, this binding prescribes an implementation of (*excl* (*HEAP* k)) assigned to a variable. Some other binding might give an implementation of polymorphic function (*excl* (*HEAP* k)), in which case the reference becomes ambiguous. In a situation such as this, the compiler chooses an implementation of a function over an assignment. A reference to a non-existent binding causes a compile-time error. This will happen when some primitive function is not implemented for a chosen data representation.

Based on the implementation bindings referenced, the compiler produces an intermediate sequential program with locations. Some primitive functions may reference bindings that prescribe incremental updates. If possible, simple execution reordering is done to preserve semantics without copying. Some optimization, such as merging locations, takes place on the intermediate code, which ultimately gets converted into code based on the implementations given in the bindings. See [27] for more details.

5. Conclusion

As noted in the introduction, the design of TL has been influenced by Polya, Standard ML with modules, and SOL. We compare the module notions of these languages with the TL notion of module. Other relevant languages include Burstall and Lampson's Pebble [4], and Cardelli and Wegner's Fun [7].

Polya is a programming language being developed at Cornell. The style of programming it encourages is described in [3,10]. Polya has a notion of module, called a *transform*, that is more general than the TL notion of module. In TL, the only kind of expression that may be implemented within a module is one that contains a single occurrence of a primitive function. As a result, new primitive functions must be declared in order to express implementations of more complicated expressions. With a Polya transform, this is unnecessary because the more complicated expressions may be implemented directly. For example, suppose we wish to implement expression (*incl* (*excl* s x) y) perhaps because for a particular representation of sets, it is more efficient to let y utilize the storage already allocated for x when x is in s. This expression cannot be implemented directly in TL. A new primitive function, say h, with the same meaning must be declared and implemented. But, by implementing h in a TL module, we effectively conceal the fact that the module really implements (*incl* (*excl* s x) y), and as a result, compilation may fail for some concrete instances when it should succeed. Suppose Q is a module of type *set int* ~ and (*h* Q *int*) is implemented, but (*incl* Q *int*) and (*excl* Q *int*) are not. Recall the definition

$$Replace = \lambda M.\ \lambda u: (set\ int)/M.\ \lambda m, n: int.\ incl\ (excl\ u\ m)\ n$$

Compiling the concrete instance (*Replace Q*) results in a compile-time error because *incl* and *excl* are not implemented for the representation in *Q*. An error results even though (*Replace Q*) is precisely function (*h Q int*) which is implemented. In contrast, Polya allows a transform to be defined that describes an implementation (transformation) of (*incl* (*excl s x*) *y*) directly.

In general, a Polya transform describes a coordinate transformation: the changing of an expression, or statement into a different form without altering its substance or intent (pg. 64 of [9]). A transform does not necessarily describe the transformation of an abstract program into a more concrete one. For example, a transform may describe the replacement of a nondeterministic action by a deterministic one.

A module in Standard ML, as proposed by MacQueen [15], is called a *structure* (actually structures and functions between them are collectively referred to as modules). An ML structure expression describes an implementation of an abstract data type (ML signature) if it appears within an **abstraction** declaration. But the purpose of ML structures, in general, is to help organize large programs, as *packages* do in other languages, whereas the scope of a TL module is limited to encapsulating implementations of abstract data types. Other aspects of TL, however, are similar to Standard ML. Each has a ramified type system, for instance. A particularly relevant paper on this subject is [14].

Mitchell and Plotkin's SOL, originally described in [19] and more recently discussed in [20], has a notion of module called a *data algebra*. The primary use of data algebras in SOL seems to be as implementations of abstract data types, which corresponds more closely to the TL notion of module. The components of a data algebra (types and functions) are accessed explicitly through an **abstype** declaraction, which ascribes names to the anonymous components. For a function that takes a data algebra as input, it is useful to statically check whether a particular data algebra, being provided as an argument, supplies the needed implementations. In practice, this might mean only ensuring that the argument has a signature that is compatible with the signature expected by the function. Signature compatibility is incorporated nicely into the type system by virtue of existential types.

In TL, only the type component of a module can be accessed explicitly (via /). There is no concept of an implementation of a primitive function belonging to a particular module, which explains why TL modules do not have existential types. Consequently, there is no concept of signature compatibility in applications of functions to modules in TL. It makes no difference whether a module to which a TL function is applied implements any of the primitive functions used in the function body. If the function is to be compiled, however, then some set of modules must supply the needed implementations.

Acknowledgments

The module notion described in this paper is the subject of the author's Ph.D. thesis, completed in 1986 under the supervision of Richard Kieburtz. While the TL notion of module has not changed, the module notation has changed as a result of the author's participation in the Polya project at Cornell. The project has been influential in the presentation of this thesis and has helped to establish a general context for the work.

Thanks to David Gries for reading earlier drafts of this manuscript and making comments that led to major improvements. Thanks also to members of the Polya project for insightful discussions.

References

1. Augustsson, L. A compiler for Lazy ML. *Proc. of 1984 ACM Conference on Lisp and Functional Programming*, August 1984, pp. 218-227.
2. Barnes, J.G.P. Programming in Ada. second edition, Addison-Wesley, International Computer Science Series, 1984.
3. Bentley, J. and Gries, D. Programming pearls. *Communications of the ACM*, vol. 30, 4, April 1987, pp. 284-289.

4. Burstall, R.M. and Lampson, B. A kernel language for abstract data types and modules. Semantics of Data Types. *Lecture Notes in Computer Science*, 173, Springer-Verlag, 1984, pp. 1-50.

5. Cardelli, L. Structural subtyping and the notion of power type. *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, 1988.

6. Cardelli, L. A Polymorphic λ-calculus with Type:Type. Technical Report 10, DEC Systems Research Center, (May 1986).

7. Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, vol. 17, 4, (Dec. 1985) pp. 471-522.

8. Cardelli, L. et al. Modula-3 Report. DEC Systems Research Center and Olivetti Research Center, (August 1988).

9. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1976.

10. Gries, D. and Prins, J.A. A new notion of encapsulation. *Proceedings of Symposium on Language Issues in Programming Environments*, SIGPLAN 20, 1985, pp. 131-139.

11. Hoare, C.A.R., Proofs of correctness of data representations, *Acta Informatica*, 1, 1972, pp. 271-281.

12. Hook, J.G. Understanding Russell - a first attempt. In Semantics of Data Types, *Lecture Notes in Computer Science*, 173, Springer-Verlag, 1984, pp. 69-85.

13. Horowitz, E. and Sahni, S. Fundamentals of data structures. Computer Science Press Inc., 1976.

14. MacQueen, D.B. Using dependent types to express modular structure. *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, 1986, pp. 277-286.

15. MacQueen, D.B. Modules for Standard ML. In Standard ML, LFCS, Department of Computer Science, University of Edinburgh, ECS-LFCS-86-2, March 1986, part III.

16. Martin-Löf, P. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, (eds. L. Cohen et al.), North-Holland, 1979, pp. 153-175.

17. Martin-Löf, P. An intuitionistic theory of types: predicative part. In *Logic Colloquium '73*, (eds. H.E. Rose and J.C. Shepherdson), North-Holland, 1975, pp. 73-118.

18. Meyer, A.R. and Reinhold, M.B. "Type" is not a type: preliminary report. *Proc. 13th Annual ACM Symposium on Principles of Programming Languages*, 1986, pp. 287-295.

19. Mitchell, J.C. and Plotkin, G.D., Abstract types have existential type, *Proc. 12th Annual ACM Symposium on Principles of Programming Languages*, 1985, pp. 37-51.

20. Mitchell, J.C. and Plotkin, G.D., Abstract types have existential type, *ACM Trans. on Programming Languages and Systems*, vol. 10, 3, (July 1988) pp. 470-502.

21. Nordström, B., Petersson, K., and Smith, J. An introduction to Martin-Löf's type theory. Department of Computer Science, University of Göteborg, March 1986.

22. Nordström, B. and Petersson, K. The semantics of data type specifications in Martin-Löf's type theory. Programming Methodology Group, Department of Computer Science, University of Göteborg, 1983.

23 Reynolds, J.C. Towards a theory of type structure. In Paris Colloquium on Programming, *Lecture Notes in Computer Science*, 19, Springer-Verlag, 1974, pp. 408-425.

24 Reynolds, J.C. Types, abstraction, and parametric polymorphism. In *IFIP Congress*, Paris, (Sept. 1983).

25. Schonberg, E. Schwartz, J.T. and Sharir, M. An automatic technique for selection of data representations in SETL programs. *ACM Trans. on Programming Languages and Systems*, 3, 2, (April 1981), pp. 126-143.

26. Stroustrup, B. The C++ Programming Language, Addison-Wesley, 1986.

27. Volpano, D.M. Software templates. Ph.D. Thesis, Department of Computer Science and Engineering, Oregon Graduate Center, Beaverton OR, (October 1986).

28. Volpano, D.M. STS - Software templates system. MCC Software Technology Program, Technical Report STP-257-87, (July 1987).

29. Wand, M. and Friedman, D. Compiling lambda expressions using continuations and factorizations. *J. Computer Languages*, 3, 1978, pp. 241-263.

30. Wand, M. Continuation-based program transformation strategies. *J. ACM*, 27, 1 (Jan. 1980), pp. 164-180.