

# Static Dependent Types for First Class Modules

Mark A. Sheldon and David K. Gifford  
Laboratory for Computer Science  
Massachusetts Institute of Technology

## Abstract

*Static dependent types* are the basis of a new type system that permits types and values to be packaged together into first class modules. Unlike other approaches to modules, static dependent types permit unrestricted access to the types and values in first class modules without sacrificing static type checking or data abstraction. Static dependent types are type safe in the presence of side effects because they rely on an *effect system* that can make deductions conventional type systems cannot. Experience with an implementation, built as an extension to the FX-87 programming language, shows that static dependent types can be used for building large systems.

## 1 Introduction

A *module* is a collection of named data types, type constructors, and values. Modules provide a means for decomposing large programs into smaller, more easily understood components. Modules typically contain tools of related functionality: for example, a matrix module might contain a matrix data type as well as subroutines for inversion, transposition, and gaussian elimination.

### 1.1 Goals

Our goals were to provide a statically type checked language with a module system supporting

- data abstraction and
- a simple facility for module linking.

Our system supports data abstraction by making module-defined types and type constructors abstract. It supports module linking by making modules first class and by allowing file input and dependent subroutines.

### Data Abstraction

Because the values in a module frequently implement one or more abstract behaviors, like matrices, stacks, trees, or hash tables, the types and type constructors defined by a

module are *abstract*. That is, the only means of manipulating values of a module-defined type are those provided by the module implementing it. This allows the module implementor to preserve internal invariants. For example, a hash table implementor may want to guarantee that hash table data structures contain an accurate count of the values currently in the hash table. This would be impossible if the user of a module could exploit knowledge of the particular details of the implementation and violate the invariant.

### Module Linking

Modules are first class if they can be passed as arguments to subroutines, returned from subroutines, and stored in data structures. Most languages, e.g., CLU [Liskov *et al.* 81] and ML [MacQueen 84], restrict program modules to be top level. The programmer must then learn a separate linking language to build a complete program from its various components. In a system with first class modules, there is no need for a separate linking language. The programmer uses the familiar notions of subroutine abstraction and application to build large programs. This observation was the motivation for Pebble [Burstall & Lampson 84] and was noted in [Boehm & Demers 86].

### 1.2 Static Dependent Types

*Static dependent types* are the basis of a new type system that permits types and values to be packaged together into first class modules. Packaging types and values together into first class modules can introduce dependencies between types and values that force types to include value expressions. For example, such a dependency arises when a value with an abstract type is packaged together with its type in a single module value. When the value is subsequently selected from the module, its type will *depend* on its parental module value. For example, if the expression (f 2) has a module type,

```
(f 2) : (modof ((t type))  
            ((y t)))
```

then (f 2).y has type (f 2).t. If a language allows value-type dependencies in modules, then a fully general type system for that language will allow type expressions to contain value expressions. A type expression that contains a value expression is called a *dependant type*. In our example the type (f 2).t is a dependant type because it contains the value expression (f 2).

The central idea of static dependant types is to restrict the side effects of value expressions that appear in type expressions. We enforce this restriction by using an *effect system* to compute the side effects of value expressions. Our use of an effect system permits us to retain static type checking for dependant types.

Previous systems of first class modules either ignore side effects, excessively restrict side effects, restrict the ability to access the types in modules, or fail to guarantee static type checking. (See Section 6 for a survey of related work.)

Section 2 shows that, to provide first class modules, free access to module types, and static type checking in the presence of side effects, a static dependent type system must observe the following design constraints:

1. Value expressions in dependent types must be independent of the store.
2. A variable's type must not depend on that variable's name or value.
3. Type equivalence, including equivalence of expressions in dependent types, must be statically computable.
4. To satisfy Constraint 1 statically, data abstraction must not interfere with static effect computation.

Constraints 1 and 4 imply that a static dependent type system must identify expressions that depend on the store or otherwise have side effects. A simple approach would be to observe that variables and `lambda` expressions are *pure*, i.e., do not cause or observe side effects. (This is Quest's approach [Cardelli 89].) In our presentation, we use a static *effect system* [Lucassen & Gifford 88] to specify and enforce somewhat looser language restrictions that meet constraints 1 and 4. This guarantees type safety in the presence of dependent types and side effects. We enforce simple scoping constraints that meet constraint 2. This prevents pathological dependencies in the type system. We use simple textual equality of value expressions to meet constraint 3. This guarantees static typability.

The system presented here meets these constraints with a minimum of restrictions. A type checker built on these rules, though with a finer-grained effect system and more language primitives, has been implemented on top of the FX-87 Interpreter [Jouvelot & Gifford 88].

### 1.3 The Base Language

The static dependent type system presented here is built on top of a simple LISP dialect called FX with a static polymorphic type and effect system in the spirit of [Lucassen & Gifford 88]. In such a language, every expression has a statically computable type and *effect*. A type describes the value an expression computes. An effect describes *how* an expression computes: it tells whether an expression reads mutable storage, writes mutable storage, and/or allocates mutable storage. Types and effects are collectively called *descriptions*.

For example, consider the polymorphic `twice` function that applies the function `f` twice on its argument:

```
(plambda ((t type) (e effect))
  (lambda ((f (subr e ((x t) t)))
    (lambda ((x t)
      (f (f x))))))
```

Formal parameters to `lambda` expressions are decorated with their types. Subroutine (`subr`) types encode the names and types of formal parameters,<sup>1</sup> the type of the return value, and the *latent effect* the subroutine will have when executed. Just as `lambda` abstracts an expression over ordinary variables of given types, `plambda` abstracts an expression over description variables of given *kinds* to yield a polymorphic value.

A polymorphic value is instantiated with the `proj` construct. For example, `twice` may be applied to the integer `square` function.

```
((proj twice int pure) square) 3)
```

Since `square` has no side effects, it is said to be *pure*.

`Twice` is a *pure* polymorphic subroutine abstracted over the type `t` of `x` (the argument of `f`) and the latent effect `e` of `f`. The type of `twice` is:

```
twice : (poly ((t type) (e effect))
          (subr pure ((f (subr e ((x t) t)))
                    (subr e ((x t) t))))
```

( $e : \tau$  means that expression  $e$  has type  $\tau$ .) Notice that `poly` types encode the kinds of the descriptions over which the value has been abstracted.

Often the existence of a `proj` and its arguments are obvious from context. Our interpreter therefore allows many `proj` expressions to be omitted. For example, a subroutine polymorphic in its argument type, when applied to an integer will be automatically projected onto `int`. In examples in the rest of the paper, we will also omit obvious projections.

### 1.4 Structure of the Paper

In the remainder of the paper:

- Section 2 develops a static dependent type system through a series of examples.
- Section 3 gives the formal syntax of the simple kernel language with an effect system, modules, and dependent types.
- Section 4 gives a formal static semantics of the kernel language, after introducing some notation, demonstrating how the above design constraints can be implemented in a practical system.
- Section 5 summarizes our initial implementation experience.
- Section 6 describes related work.
- Section 7 makes concluding remarks.

## 2 Modules and Static Dependent Types

Our static dependent type system consists of three parts: facilities for creating and using modules, dependent subroutines, and support for separate compilation. In this section, we shall discuss each of these parts separately, relying heavily on simple examples. Taken together, these features represent a powerful and flexible system that allows a language

<sup>1</sup>Subroutine types in [Lucassen & Gifford 88] do not encode the formal names, however, this will be necessary for dependent subroutines below.

to be its own linking language. Our static dependent type system allows our language, unlike previous languages, to provide this functionality and guarantee static type checking.

## 2.1 Modules

### 2.1.1 Creating Modules

One builds a module by specifying a set of abstraction definitions and a set of field definitions:

- An *abstraction definition* is a binding of the form  $(I K D)$ . The identifier  $I$  is the abstraction name. The kind expression  $K$  tells whether the abstraction is a type or a type constructor. The description  $D$  is the abstraction's representation. The abstraction definitions are mutually recursive.<sup>2</sup>
- A *field definition* is of the form  $(I T E)$ . The identifier  $I$  is the field name.  $T$  is the type of the value exported by the field.  $E$  is an expression that is the field's definition. Field definitions are mutually recursive and may use the abstractions defined in the module.

For example, suppose  $M$  is the following expression, creating a module with one abstract type  $t$  and one field  $x$  of the abstract type:

```
(mod ((t type int))
      ((x t (up-t 0))))
```

(Up-t converts from the representation to the abstract type.)

The types of the values are *not* in the scope of the field names. This prevents the type of a field from referring to that field in its type as in the following binding:

```
(mod
  ...
  (f f.foo (mod (... (foo ...) ...) ...))
  ...)
```

If such a binding were legal, kind checking  $f.foo$  would involve knowing the type of  $f$  which involves kind checking  $f.foo$ . This pathological dependency is the motivation for Design Constraint 2.<sup>3</sup>

In a language with effect masking like FX-87 [Gifford *et al.* 87], one must be careful that effects are not masked as a result of the conversion between abstract types and their concrete representations. This would allow an effect loophole. Unfortunately, the coarse-grained effect system of this paper is too simple to exhibit this problem. For a more complete discussion of this issue and for restrictions and an effect masking rule in a finer-grained effect system, see [Sheldon 90]. This is Design Constraint 4.

### 2.1.2 Module Types

In our system, modules are first class values and thus have types. By way of contrast, ML [MacQueen 84] has a separate signature mechanism for describing modules, because modules are not first class values.

<sup>2</sup>Description functions are fully supported. Recursive calls are treated abstractly and are not  $\beta$ -reduced, thus preserving static decidability.

<sup>3</sup>Every example we have constructed to exhibit this pathology is, in fact, non-typeable and non-sensical; so the restriction seems quite reasonable.

Module types encode the kinds of the abstractions they implement and the types of the values they contain. For example, if  $M$  is the simple module of the previous section,

```
M : (modof ((t type))
           ((x t)))
```

For subsequent examples, we will use a more practical module. Let `pair-type` be the type of a module implementing mutable pairs:

```
(modof
  ((pairof (dfunc (type type) type)))
  ((mk-pair (poly ((t1 type) (t2 type))
                  (subr alloc ((fst t1) (snd t2))
                              (pairof t1 t2))))
  (fst      (poly ((t1 type) (t2 type))
                  (subr read ((p (pairof t1 t2))
                              t1))))
  (snd      (poly ((t1 type) (t2 type))
                  (subr read ((p (pairof t1 t2))
                              t2))))
  (set-fst! (poly ((t1 type) (t2 type))
                  (subr write ((p (pairof t1 t2))
                              (x t1))
                              unit))))
  (set-snd! (poly ((t1 type) (t2 type))
                  (subr write ((p (pairof t1 t2))
                              (x t2))
                              unit))))))
```

An implementation of a module of this type must define a `pairof` type constructor (a description function mapping two types to a type), as well as polymorphic subroutines for pair creation, destructuring, and mutation. Creating a pair has an allocation effect, accessing pair components has a read effect, mutating pair components has a write effect.

### 2.1.3 Extracting Descriptions from Modules

Descriptions can be extracted from modules just as values can. For example, if

```
pair-impl : pair-type
```

then one could extract the abstraction by using the familiar dot notation:

```
pair-impl.pairof :: (dfunc (type type) type)
```

(The notation  $\delta :: \kappa$  means description  $\delta$  has kind  $\kappa$ .)

The dot notation is actually a syntactic shorthand for the `select` form. Thus one could also write:

```
(select pair-impl pairof)
```

Projecting and calling `mk-pair` to create a pair of integers would produce a result of type:

```
(pair-impl.pairof int int)
```

Notice that a description that selects an abstraction from a module contains, or depends upon, a value expression. For this reason, these descriptions are called *dependent descriptions*.<sup>4</sup>

<sup>4</sup>Elsewhere in the literature, this form is sometimes called a *witness* operator.

An abstraction selected from a module is *not* convertible to its representation. If it were, it would not be abstract: a user of the module could rely on the representation to violate invariants required by the module implementation. Thus, if *M* is the first module example, (`select M t`) is not the same as `int`. This contrasts with Pebble [Burstall & Lampson 84].

If type checking is to be decidable, then description equivalence must also be decidable. This is the motivation for Design Constraint 3. The approach taken here is, perhaps, the simplest one: Two `select` forms are equivalent iff they are textually identical.<sup>5</sup> Thus, there is no need for the type checker to evaluate the module expression in a `select`, which might not terminate. The question of description equivalence is more fully addressed in Section 4.4 below.

If the implementation of an abstraction used in a program could change dynamically while the syntactic type of a value remained the same, then the language would no longer be type safe. (See Section 2.2 for an example of this problem.) This is the motivation for Design Constraint 1, that expressions in dependent types must be independent of the store. If this is true, then two expressions with the same abstract type must have come from the implementation of the same module (or from syntactically identical implementations in the same file).

Requiring that the module expression in a `select` be `pure` is too restrictive. Only if it *reads* mutable storage that another expression may alter is it possible for an implementation to change dynamically. Thus, we meet this constraint by requiring that the static effect of the module expression not contain `read` effects. (Quest is much more restrictive, requiring that the module expression be an identifier.)

The ability to articulate and enforce this effect restriction allows this system to do what others do not: to combine first class modules, dependent types, and free use of side effects in a type safe language. The inability to express this property is what leads designers to restrict the use of, and selection from, modules.

(See the description of the `select` kind rule in section 4.2 below for the precise conditions necessary to preserve static type checking in the presence of dependent types.)

### 2.1.4 Extracting Values from Modules

The `with` expression selects values from modules. Inside a `with` expression, the programmer may refer directly to the value *and* abstraction names defined by a module. The following `with` expression applies the `mk-pair` subroutine in a pair implementation.

```
(with pair-impl
  (mk-pair 0 #\a)) : (pair-impl.pair of int char)
```

The type of the value returned from a `with` expression may contain references to the identifiers exported by the module. Abstraction names are replaced by appropriate selections, and field names are replaced by `with` expressions. Thus within the `with` body, the type of the value created is `(pair of int char)`; after it is exported, its type is as above.<sup>6</sup> SOL [Mitchell & Plotkin 88] and Quest do not allow this because they do not have dependent types.

<sup>5</sup>We shall see below that substitutions are performed on dependent types.

<sup>6</sup>The bindings made by `with` are *opaque*. See the discussion of `let` in the next section for a definition and explanation of opacity.

We also overload the conventional dot notation. In a description, it is sugar for `select`. In a value expression, it is sugar for a `with` expression with an identifier in its body: `M.x` is the same as `(with M x)`.

## 2.2 Dependent Subroutines

A *dependent subroutine* is one in which either the return type depends on one of the formal parameters or the type of a parameter depends on an earlier parameter. Dependent subroutines are the basis of module linking. (Functors in ML are a restricted form of dependent subroutines.)

Dependent subroutines are useful for writing programs that take both an implementation of an abstraction and an object of the abstract type. Suppose `pair-example` is a subroutine that takes an implementation of pairs satisfying the `pair-type` interface and a pair created with that implementation. It then constructs a new pair:

```
(lambda ((m pair-type) (p (m.pair of int char)))
  (with m
    (mk-pair (fst p) (char->int (snd p)))))
: (subr (maxeff alloc read)
  ((m pair-type) (p (m.pair of int char)))
  (m.pair of int int))
```

Notice the use of implicit projection [Gifford *et al.* 87]. `Char->int` returns the integer equivalent of a character.

This program cannot be written in CLU or ML because their modules are not first class. In SOL and Quest a value of abstract type may not be returned from a scope in which the abstraction is opened.

Suppose `pair-impl` : `pair-type`. Then, applying this dependent subroutine:

```
(pair-example pair-impl (pair-impl.mk-pair 3 #\a))
: (pair-impl.pair of int int)
```

Notice that `pair-impl` was substituted for the formal parameter `m` in the type of `pair-example`.

Dependent subroutines allow us to render the motivation for Design Constraint 1 more explicit. Suppose that `mk-pair-impl` is a subroutine that takes a string describing which of several possible pair implementations the caller desires. Then suppose that `M` stands for the expression

```
(mk-pair-impl (read-string-from-keyboard))
```

This expression returns a pair module according to the selection made by the user at run time. If effects were not considered, the following program would type check:

```
((with M fst)
  ((with M mk-pair) 0 1))
```

`((select M pair of) int int)` is the syntactic type expected by the subroutine `(with M fst)`. But the above program reads twice from the keyboard. The user may enter different selections each time, and `mk-pair-impl` may return incompatible pair implementations for the two cases. This would lead to a run time type error. Our system avoids this problem by recognizing that the `read-string-from-keyboard` subroutine has a `read` effect. Thus the `select` expression used in the type of the pair is illegal.

It may, however, be useful to allow the user to select an implementation dynamically. This is an important consideration in the design of the `let` construct:

```
(let ((pair-impl (mk-pair-impl "lambda-based")))
  (pair-example
   pair-impl (pair-impl.mk-pair 3 #\a))
  : ((select (mk-pair-impl "lambda-based") pair-of)
    int int))
```

Notice that `let` has the same sort of substitution properties as application. In fact, our semantics for `let` are derived from that of subroutine application. Therefore, the bindings of a `let` expression are *opaque*: Even if two identifiers are bound to textually identical expressions, `select` descriptions from the two variables are *not* interconvertible. This makes sense because the programmer who binds something to two different names has two different purposes in mind. But more significantly, it means that a module computed using side effects may be used locally:

```
(let ((pair-impl (mk-pair-impl
                  (read-string-from-keyboard))))
  (pair-impl.fst (pair-impl.mk-pair 0 1)))
: int
```

The effect constraint on `select` implies that such a `let` may not export a pair created with its pair implementation: After the substitution of the definition of `pair-impl` into the type, we would get an illegal `select` containing an expression with a `read` effect.

This means that in the presence of side effects that may undermine type safety, our system enforces exactly the same constraints as the existential type system in SOL. But our system enforces this *only when necessary*.

### 2.3 Linking and Separate Compilation

Because our system has both first class modules and dependent subroutines, it is its own linking language. All that remains is to provide a means for getting values out of a file system. The `input` form does just that. The type of an `input` expression is the type of the expression in the indicated file (or of the compiled code if the expression names an object file). The file named in an `input` form *must* exist at compile time so that its type can be known. No free variables are allowed in such a separate file. A real implementation would have a standard library in scope for such files by implicitly enclosing the code in all files in a `(with library ...)`. A trivial program linking in a pair implementation might look like this:

```
((lambda ((pair-impl pair-type))
  (pair-example pair-impl
   (pair-impl.mk-pair 0 #\a)))
(input "FX;IMPL;PAIROF.FX"))
: ((select (input "FX;IMPL;PAIROF.FX") pair-of)
  int int)
```

Our implementation actually builds the system library module out of its component modules in this way.

For this to be type-safe, the implementation represented by the `input` form must not change. Thus, we must provide the illusion that the file system is immutable. Our current implementation attaches the last update time to the `input` node in the parse tree and uses this to compare types that select abstractions from files. For a detailed discussion of this issue, see [Sheldon 90].

## 3 Syntax of a Kernel Language

Figure 3 contains a BNF grammar for the language of this paper, a simple polymorphic, typed, lisp dialect in the spirit of FX-87 [Gifford *et al.* 87, Lucassen & Gifford 88]. Our language allows side effects and first class subroutines.

The grammar does not describe the dot notation for extracting abstractions and values from modules. These are syntactic sugar eliminated by the parser.

$K$	=	effect type (dfunc ( $K \dots$ ) $K$ )	Kinds ( $\kappa$ ) description function
$F$	::=	write read alloc (maxeff $F \dots$ )	Side effects ( $\epsilon$ ) mutate observe allocate combination
$T$	::=	(subr $F$ (( $I T$ )...) $T$ ) (poly (( $I K$ )...) $T$ ) (modof (( $I K$ )...) (( $I T$ )...))	Types ( $\tau$ ) dependent subroutine polymorphic module (interface)
$D$	::=	$F$ $T$ $I$ (dlambda (( $I K$ )...) $D$ ) ( $D D \dots$ ) (select $E I$ )	Descriptions ( $\delta$ ) variable abstraction application description selection
$E$	::=	$I$ (lambda (( $I T$ )...) $E$ ) ( $E E \dots$ ) (plambda (( $I K$ )...) $E$ ) (proj $E D \dots$ ) (mod (( $I K D$ )...) (( $I T E$ )...)) (with $E E$ ) (input <i>file-name</i> )	Expressions ( $e$ ) variable lambda abstraction application polymorphic abstraction projection module creation value selection file input

Figure 1: Grammar for a kernel language.

In the effect domain, expressions can `write`, `read` or `allocate` (`alloc`) memory. Combined effects are introduced by `maxeff`. The effect of a pure expression is written (`maxeff`), though we allow `pure` as a synonym for the empty `maxeff`.

## 4 Static Semantics

The static semantics, in addition to the usual function of providing a way to check the well-formedness and type/effect safety of programs, also guarantees that at run time, any two values of the same abstract type were created by the same implementation of the abstract type.

This section begins by explaining the notation used in the rest of the section. Then, it provides a static semantics for the language whose syntax was given above: kind inference rules, type and effect inference rules, and description inclusion rules. For the sake of brevity, the rules that are the same as for the second-order polymorphic lambda calculus [McCracken 79] are omitted. For a complete set of rules, see [Sheldon 90].

## 4.1 Notation

The use of  $:$  to denote the *type of* relation and of  $::$  to denote the *kind of* relation is familiar. We will similarly use  $!$  to denote the *effect of* relation.

The inference rules given below are written as deductions with premises above the horizontal line, conclusions below. We will make use of an environment,  $TK$ , which is the type and kind assignment function mapping variables to their type or kind.

The notation  $TK[x : y]$  denotes functional extension of  $TK$ , i.e., it denotes the new function identical to  $TK$  except that it maps  $x$  to the type  $y$ . We will similarly use  $TK[x :: y]$ . We will assume that all input programs are  $\alpha$ -renamed so that for any such extension  $x \notin \text{Domain}(TK)$ . This is not a restriction on programs, as  $\alpha$ -renaming is done by the parser.<sup>7</sup>  $[\![_{i=1}^n x_i : y_i]$  is an abbreviation for  $[x_1 : y_1] \dots [x_n : y_n]$ , and similarly for  $::$ . Environment extension is left-associative.

The notation  $[x/y]\delta$  denotes the result of substituting  $x$  for all occurrences of  $y$  in  $\delta$ . As above,  $[\![_{i=1}^n x_i/y_i]$  is an abbreviation for  $[x_n/y_n] \dots [x_1/y_1]$ . Substitution is right associative.

## 4.2 Kind Inference Rules

In order to determine that descriptions are well-formed, we use *kinds* [McCracken 79]. A description is well-formed iff it has a kind under these kind inference rules. The rules which differ from the polymorphic, typed lambda calculus are those for module signature (**modof**), extraction of module abstraction (**select**), and dependent subroutine (**subr**).

### modof

The **modof** rule says that the types of the value bindings must be well-formed when the abstraction names are available to them. Because the semantics of abstract effects are unclear, the current design requires that abstractions be types or type constructors. We will see below why the types in a **modof** may not refer to the  $idv_i$ .

$$\frac{TK[\![_{i=1}^n ida_i :: \kappa_i] \vdash \tau_j :: \text{type} \quad (1 \leq j \leq m)}{\text{FinallyType}(\kappa_i) \quad (1 \leq i \leq n)} \quad TK \vdash (\text{modof} \ ((ida_1 \ \kappa_1) \dots (ida_n \ \kappa_n)) \ ((idv_1 \ \tau_1) \dots (idv_m \ \tau_m)))$$

FinallyType has the recursive definition:

$$\begin{aligned} \text{FinallyType}(\text{type}) &= \text{true} \\ \text{FinallyType}(\text{dfunc}(\kappa_1 \dots \kappa)) &= \text{FinallyType}(\kappa) \end{aligned}$$

### select

One can only **select** a description from a module. The **select** rule therefore requires that the expression it contains have a **modof** type.

$$\frac{TK \vdash e \quad \text{read} \ \not\sqsubseteq \ \epsilon}{TK \vdash (\text{select} \ e \ \text{ida}_i) :: \kappa_i}$$

<sup>7</sup>The body of a **with** form cannot be  $\alpha$ -renamed until its module expression is type checked. So in this case, some parsing is interleaved with type checking.

( $\sqsubseteq$  is the description inclusion relation defined below.)

Languages without dependent types have the property that, though the type-checking rules often require kind deductions, the kind-checking rules never require type deductions. The **select** kind rule violates this principle. Will some programs create infinite chains of deductions?

Expressions written by programmers are always finite and cannot contain cycles. As long as type-checking the expression in a **select** does not require information outside of the expression itself, the system will operate on smaller and smaller expressions and eventually terminate. But what information from outside of an expression is needed to type-check the expression? Only the types of the free value variables (in  $TK$ ) are needed. Provided that these types are already known and kind-checked, the deduction will be finite.

Requiring that free value variables in a description already have their types kind-checked and in  $TK$  ensures that the language meets Design Constraint 2, that the type of a variable not depend on that variable. For example, a field name in a module construct may not appear free in a **select** in its own type (as in the example in section 2.1.1). This is because its type is not yet kind-checked and in  $TK$ .

### subr

Dependent subroutine types are binding constructs. Each formal name is available to the types of all later formals and to the return type. This allows one to abstract code over an implementation of a type and to use values of that type freely.

In accordance with Design Constraint 2, no formal name is available to its own type.

$$\frac{TK \vdash \epsilon \quad :: \ \text{effect} \quad TK[\![_{i=1}^j id_i : \tau_i] \vdash \tau_{j+1} \quad :: \ \text{type} \quad (0 \leq j \leq n)}{TK \vdash (\text{subr} \ \epsilon \ ((id_1 \ \tau_1) \dots (id_n \ \tau_n)) \ \tau_{n+1}) \quad :: \ \text{type}}$$

## 4.3 Type and Effect Inference Rules

The type and effect inference rules provide a way of verifying that an expression is well-formed: An expression is well-formed iff it has a type and effect under these rules. The expressions whose rules differ from the ordinary polymorphic lambda calculus are: **lambda**, application, **let**, **mod**, **with**, and **input**.

### lambda

Dependent subroutines require a new typing rule for **lambda** expressions so that any formal name may be used in the type expressions of any following formals and in the return type. In accordance with Design Constraint 2, no formal name is available to its own type.

$$\frac{TK[\![_{i=1}^j id_i : \tau_i] \vdash \tau_{j+1} \quad :: \ \text{type} \quad (0 \leq j \leq n) \quad TK[\![_{i=1}^n id_i : \tau_i] \vdash e \quad : \ \tau_{n+1} \ ! \ \epsilon}{TK \vdash (\text{lambda} \ ((id_1 \ \tau_1) \dots (id_n \ \tau_n)) \ e) \quad : \ (\text{subr} \ \epsilon \ ((id_1 \ \tau_1) \dots (id_n \ \tau_n)) \ \tau_{n+1}) \quad ! \ \text{pure}}$$

### application

When a dependent subroutine is applied, the formal names appearing in the return type must be replaced by the argument expressions.

$$\frac{\begin{array}{l} \text{TK} \vdash e \quad : \quad (\text{subr } \epsilon ((id_1 \tau_1) \dots (id_n \tau_n)) \tau) \\ \quad \quad \quad ! \quad \quad \quad \epsilon \\ \text{TK} \vdash e_i \quad : \quad [_{j=1}^{i-1} e_j / id_j] \tau_i ! \epsilon_i \quad (1 \leq i \leq n) \\ \text{TK} \vdash [_{i=1}^n e_i / id_i] \tau \quad :: \quad \text{type} \end{array}}{\text{TK} \vdash (e \ e_1 \dots) : [_{i=1}^n e_i / id_i] \tau ! (\text{maxeff } \epsilon \ \epsilon_1 \dots)}$$

Verifying the kind of the result type after substitution is a compact way to check that any expression actually substituted into the result type has an effect commensurate with Design Constraint 1. Forcing the result type to have kind **type** means that, in particular, all dependent descriptions must be well-kinded. This in turn means that the expressions in them will not have **read** effects. The rule would be much more complicated if it applied the effect constraint directly to the argument expressions actually substituted into dependent descriptions in the result type.

### let

The **let** rule must change in the same way as the application rule.

$$\frac{\begin{array}{l} \text{TK} \vdash e_i \quad : \quad \tau_i ! \epsilon_i \quad (1 \leq i \leq n) \\ \text{TK} [_{i=1}^n id_i : \tau_i] \vdash e \quad : \quad \tau ! \epsilon \\ \text{TK} \vdash [_{i=1}^n e_i / id_i] \tau \quad :: \quad \text{type} \end{array}}{\text{TK} \vdash (\text{let } ((id_1 \ e_1) \dots) \ e) : [_{i=1}^n e_i / id_i] \tau ! (\text{maxeff } \epsilon \ \epsilon_1 \dots)}$$

The fact that **let** bindings are opaque in the way described above in Section 2.2 results from the way substitution is used. Substitutions are carried out only *after* the result type of the **let** body has been computed. This means that within the **let** body, selections from **let**-bound variables are not convertible with selections from the variables' definitions. This is a consequence of making the semantics of **let** follow the semantics of an applied **lambda**. As shown in Section 2.2, this arrangement allows a module-producing computation with **read** effects to be used as long as the corresponding identifier is not free in the type of the **let** body. This means that our system enforces the restrictions of SOL *only when necessary* to guarantee type safety.

### mod

$$\frac{\begin{array}{l} \text{TK}_1 \quad = \quad \text{TK} [_{i=1}^n id_i :: \kappa_i] \\ \text{TK}_1 \vdash \delta_i \quad :: \quad \kappa_i \quad (1 \leq i \leq n) \\ \text{FinallyType}(\kappa_i) \quad (1 \leq i \leq n) \\ \text{TK}_1 \vdash \tau_i \quad :: \quad \text{type} \quad (1 \leq i \leq m) \\ \text{TK}_2 \quad = \quad \text{TK}_1 [_{i=1}^n \text{up-} id_i \\ \quad \quad \quad : \text{UpType}(id_i, \kappa_i, \delta_i)] \\ \text{TK}_3 \quad = \quad \text{TK}_2 [_{i=1}^n \text{down-} id_i \\ \quad \quad \quad : \text{DownType}(id_i, \kappa_i, \delta_i)] \\ \text{TK}_3 [_{i=1}^m idv_i : \tau_i] \vdash e_j : \tau_j \quad (1 \leq j \leq m) \end{array}}{\text{TK} \vdash (\text{mod } ((ida_1 \ \kappa_1 \ \delta_1) \dots (ida_n \ \kappa_n \ \delta_n)) \\ ((idv_1 \ \tau_1 \ e_1) \dots (idv_m \ \tau_m \ e_m))) \\ : (\text{modof } ((ida_1 \ \kappa_1) \dots (ida_n \ \kappa_n)) \\ ((idv_1 \ \tau_1) \dots (idv_m \ \tau_m))) \\ ! (\text{maxeff } \epsilon_1 \dots \epsilon_m)}$$

The definitions of **UpType** and **DownType** are given below.

The **mod** typing rule embodies many important design decisions and is the most complicated rule of the type system. It thus deserves a detailed description.

The use of  $\text{TK}_1$  in kind checking the representations allows the representations to be mutually recursive. The recursion here is *opaque* in the sense that recursive references in the descriptions are not substituted away. This supports the use of abstractions internally to the module by forcing the programmer to be aware of every conversion between an abstract and concrete type. It also means that recursive description functions do not violate static type checking: recursive calls are not  $\beta$ -reduced.

In accordance with Design Constraint 2 the interface types are kind checked in an environment where the abstraction names are available, but the field names are not.

There is a rich design space of mechanisms for converting between an abstract type and its concrete representation. See [Sheldon 90] for a discussion of the issues involved. The option settled on in this design is to provide subroutines bound to names derived from the abstraction names. The subroutines are polymorphic if the abstraction is higher kinded. (See the definitions of **UpType** and **DownType** below.) With such a scheme, the programmer only converts between the abstract and concrete types when necessary, taking advantage of the abstraction internally whenever possible. Where additional special forms would complicate the static semantics with more rules, subroutines do not. The semantics of these subroutines are very simple: they are identity subroutines.

$$\begin{aligned} \text{UpType}(\delta_a, \text{type}, \delta_r) &= (\text{subr pure } ((x \ \delta_r)) \ \delta_a) \\ \text{UpType}(\delta_a, (\text{dfunc } (\kappa_1 \dots \kappa_n) \ \kappa_{n+1}), \delta_r) &= \\ &(\text{poly } ((x_1 \ \kappa_1) \dots (x_n \ \kappa_n)) \\ &\text{UpType}(\delta_a \ x_1 \dots x_n, \kappa_{n+1}, (\delta_r \ x_1 \dots x_n))) \end{aligned}$$

$$\text{DownType}(\delta_a, \kappa, \delta_r) = \text{UpType}(\delta_r, \kappa, \delta_a)$$

The variables  $x$  and the  $x_i$  introduced as formals are fresh identifiers.

### with

The expression in the body of a **with** form is evaluated in an environment in which the identifiers exported by the module expression are bound to their respective abstractions and values.

$$\frac{\begin{array}{l} \text{TK} \vdash e_m \quad : \quad (\text{modof } ((ida_1 \ \kappa_1) \dots (ida_n \ \kappa_n)) \\ \quad \quad \quad ((idv_1 \ \tau_1) \dots (idv_m \ \tau_m))) \\ \quad \quad \quad ! \quad \quad \quad \epsilon_m \\ \text{TK}_1 \quad = \quad \text{TK} [_{i=1}^n id_{a_i} :: \kappa_i] \\ \text{TK}_2 \quad = \quad \text{TK}_1 [_{j=1}^m idv_j : \tau_j] \\ \text{TK}_2 \vdash e \quad : \quad \tau ! \epsilon \\ \quad \quad \quad \tau' \quad = \quad [_{j=1}^m (\text{with } e_m \ idv_j) / idv_j] \tau \\ \quad \quad \quad \tau'' \quad = \quad [_{i=1}^n (\text{select } e_m \ id_{a_i}) / id_{a_i}] \tau' \\ \text{TK} \vdash \tau'' \quad :: \quad \text{type} \end{array}}{\text{TK} \vdash (\text{with } e_m \ e) \quad : \quad \tau'' ! (\text{maxeff } \epsilon_m \ \epsilon)}$$

Verifying the kind of the result type after substitution is a compact way to check that any expression actually substituted into the result type has an effect commensurate with Design Constraint 1.

**input**

The `input` form allows programs to be split into multiple files.

$$\frac{\phi \vdash \text{FS}(\text{string-literal}) : \tau ! \epsilon}{\text{TK} \vdash (\text{input string-literal}) : \tau ! \epsilon}$$

$\phi$  represents the empty type and kind environment. FS is the mapping from string literals to FX expressions represented by the file system. In order for this system to be type-safe, FS must be a *function*: the file system must be immutable. This is actually not a bad restriction: Chapter 3 of [Sheldon 90] demonstrates how this can be implemented on a standard file system. Our implementation tags `input` expression with the last file write date in universal time.

**4.4 Description Inclusion**

Two descriptions are the same iff they can be proved equivalent under the description equivalence relation. We present here only the the `modof` and `select` rules.

The only use the above rules make of description inclusion is to verify that expressions in `select` descriptions do not have `read` effects. There are many implicit appeals to a notion of description equivalence when, for example, the deduced argument types of a subroutine application must be equivalent to the corresponding subroutine parameter types.

The rules can be thought of as defining a reduction semantics for the simply typed lambda calculus with  $\alpha$ ,  $\beta$ , and  $\eta$  conversion. Thus, all descriptions have normal forms modulo  $\alpha$  conversion.

**modof**

Since modules often implement abstract behaviors, it is usually useful to ignore field order in `modof` types. Thus two `modof` types that differ only in the order of abstractions and fields are equivalent:

$$\frac{\begin{array}{l} \pi \text{ is a permutation on } [1, n] \\ \pi' \text{ is a permutation on } [1, m] \end{array}}{\begin{array}{l} (\text{modof } ((ida_1 \ \kappa_1) \dots (ida_n \ \kappa_n)) \\ ((idv_1 \ \tau_1) \dots (idv_m \ \tau_m))) \\ \equiv \\ (\text{modof } ((ida_{\pi(1)} \ \kappa_{\pi(1)}) \dots (ida_{\pi(n)} \ \kappa_{\pi(n)}) \\ ((idv_{\pi'(1)} \ \tau_{\pi'(1)}) \dots (idv_{\pi'(m)} \ \tau_{\pi'(m)}))) \end{array}}$$

Subtyping on `modofs` is also useful. To get a subtype of a `modof` type, one may add new fields and/or take subtypes of the fields:

$$\frac{\begin{array}{l} \tau'_i \sqsubseteq \tau_i \quad (1 \leq i \leq n) \\ m \leq m' \\ n \leq n' \end{array}}{\begin{array}{l} (\text{modof } ((ida_1 \ \kappa_1) \dots (ida_{n'} \ \kappa_{n'}) \dots) \\ ((idv_1 \ \tau_1) \dots (idv_{m'} \ \tau_{m'} \dots))) \\ \sqsubseteq \\ (\text{modof } ((ida_1 \ \kappa_1) \dots (ida_n \ \kappa_n)) \\ ((idv_1 \ \tau_1) \dots (idv_m \ \tau_m))) \end{array}}$$

**select**

There is no inclusion rule on `select` descriptions, only an equivalence rule. This way if two expressions have compatible dependent types, then they were created by using the

same implementation of the module. If this were not the case, then a programmer could, for example, pass a pair created with one implementation to the `snd` operation of another implementation. The other implementation might use an incompatible representation of pairs, and a type error would occur at run time.

Because dependent descriptions contain value expressions, their equivalence rule requires an equivalence rule for the expressions appearing in `selects`.

$$\frac{e \equiv_v e'}{(\text{select } e \ id) \equiv (\text{select } e' \ id)}$$

There are many choices for the definition of  $\equiv_v$ : Equivalence on expressions could be defined as equivalence on the values they compute (given a definition of equivalence on values). But this would require arbitrary, possibly non-terminating computation in the type-checker. Equivalence could be based on a simpler sort of evaluation that involves  $\alpha$  and  $\eta$  conversion, and limited  $\beta$  substitutions (just in `let` expressions, say). But this requires the programmer to understand yet a third sort of reduction (the other two being the reduction of standard value expression evaluation and the reduction implied by the inclusion rules on descriptions). The simplest equivalence relation is textual equality. This will suffice to satisfy Design Constraint 3. A formal definition of textual equality is straight-forward and tedious. It is therefore omitted.

Note: substitutions into descriptions in the kind and type inference rules apply to the expression in a `select`. Thus if the body of a `let` expression has a type `(select x y)` for an `x` bound to `e` by the `let`, then the type of the `let` is `(select e y)`. Thus descriptions never contain identifiers unbound in the current scope.

**5 Implementation Experience and Future Work**

We have implemented a static dependent type system as part of the FX-87 Interpreter [Jouvelot & Gifford 88]. Our experience shows that such a system is powerful enough to allow a language to be its own linking language. In particular, we built a kernel FX-87 runtime system in several modules in separate files. Then, a program, itself written in FX-87 was used to link the modules into one large `fx` module.

We are currently working on a module system with several feature that make modules easier to use: local bindings in modules, transparent (i.e. not abstract) description bindings, and simple means of combining modules.

Substitutions in our implementation were very time consuming, but the additional time was localized to programs using the new constructs: old programs did not take appreciatively longer to type-check. It may be possible to increase the implementation's efficiency by exploiting sharing in the type checker the way ML does.

For a more detailed description of implementation issues, see [Sheldon 90].

**6 Related Work**

Most previous systems, like CLU and ML, do not allow modules to be first class. They therefore require a second linking language, when the usual notions of subroutine abstraction, application, and conditionalization, provide ample power.

Other systems of first class modules and dependent types can be divided into two groups: systems that do not guarantee static type checking and systems with overly severe restrictions on dependent types or use of side effects.

The strong existential type system of Pebble [Burstall & Lampson 84] does not guarantee static type checking. [Hook & Howe 86] shows that *no* system of strong existential types and polymorphism is statically typeable. Moreover, the module types in a strong existential type system are not abstract. This is why Pebble resorts to a password system to implement type abstraction. Pebble also does not have side effects.

Quest [Cardelli 89] is a static system with dependent types, but it does not have dependent subroutines and does not allow values whose types depend on a (recursive) `let`-bound variable to be exported from the scope of the `let`.

Russell [Boehm *et al.* 80] takes an approach quite similar to our own. Their separate compilation mechanisms are especially similar [Boehm & Demers 86]. Unfortunately, Russell suffers from a weak effect system: Since Russell detects an impure expression by the presence of reference types in its type, Russell cannot distinguish placing a reference in a data structure from mutating and extracting the value. Our effect system can detect `read` effects specifically.

Another problem in Russell results from the failure to separate the expression and type domains. That is, restrictions on types intended to guarantee type safety apply as well subroutines.

Russell's *import rule* is therefore overly restrictive. It says that operations, including subroutines and modules, may not contain free variables of a reference type. This outlawed global mutable variables. If dependent types were separated from the expression domain, then such a restriction could be applied only to those forms.

Russell's *export rule* forbids values whose types depend on a (recursive) `let`-bound variable to be exported from the scope of the `let`. This is due to the lack of a substitution rule for recursive bindings. Our rule for this case arises from the `with` and `mod` rules.

Because Russell assumes only one abstraction *per* module, it does not distinguish between types and values at all. But value expressions denoting types are compared textually to test for equivalence. Application and dependent subroutines work very much as they do here.

Because Russell programs are not automatically  $\alpha$ -renamed, programmers may not redeclare an identifier in a `let` expression or an identifier that appears free in the type of some free variable of the `let` body.

## 7 Conclusion

Static dependent types support a first class module system that is more flexible than other statically typed systems. This flexibility allows the user to use the same organizational principles to write programs and to link multiple modules together into a large system: programs are like data. Minimal static effect constraints guarantee that the type system is sound in the presence of side effects. This system has been implemented with finer-grained effects and more language primitives on top of the FX-87 Interpreter [Jouvelot & Gifford 88].

## Acknowledgments

The kernel idea of this paper, that there should be a static semantics for dependent types is due to Mark Reinhold. He worked out a scheme in a language without side effects and higher order descriptions. He also collaborated in the extension of the idea to include side effects and higher order, mutually recursive descriptions.

In addition, we are indebted for technical contributions and comments to the conference referees, Michael Blair, Pierre Jouvelot, James O'Toole, Jonathan Rees and Franklyn Turbak.

This research was funded by the Defense Advanced Research Projects Agency of the U.S. Department of Defense, contract number N00014-83-K-0125.

## References

- [Boehm & Demers 86] Hans-Juergen Boehm and Alan Demers. Implementing russell. In *Symposium on Compiler Construction*, pages 186–195, ACM Sigplan, June 1986.
- [Boehm *et al.* 80] H. Boehm, A. Demers, and J. Donahue. *An Informal Description of Russell*. Technical Report TR80-430, Cornell University, Department of Computer Science, 1980.
- [Burstall & Lampson 84] R. Burstall and B. W. Lampson. A kernel language for abstract data types and modules. In *Semantics of Data Types*, Springer-Verlag, 1984.
- [Cardelli 89] Luca Cardelli. Typeful programming. In *IFIP Advanced Seminar on Formal Description of Programming Concepts*, 1989.
- [Gifford *et al.* 87] David K. Gifford, Pierre Jouvelot, John M. Lucassen, and Mark A. Sheldon. *FX-87 Reference Manual*. Technical Report MIT/LCS/TR-407, M.I.T. Laboratory for Computer Science, September 1987.
- [Hook & Howe 86] James G. Hook and Douglas J. Howe. *Impredicative Strong Existential Equivalent to Type:Type*. Technical Report TR 86-760, Department of Computer Science, Cornell University, Ithaca, New York, June 1986.
- [Jouvelot & Gifford 88] Pierre Jouvelot and David K. Gifford. The FX-87 interpreter. In *Proceedings of the 2nd Int'l. Conf. on Comp. Lang.*, Miami, Florida, October 1988.
- [Liskov *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, and Alan Snyder. *CLU Reference Manual. Lect. Notes in Computer Sci.*, Springer-Verlag, 1981.
- [Lucassen & Gifford 88] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages*, ACM, 1988.

- [MacQueen 84] David MacQueen. Modules for standard ML. In *Proceedings ACM Symposium on Lisp and Functional Programming*, 1984.
- [McCracken 79] Nancy Jean McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure*. Ph.D. thesis, Syracuse University, 1979.
- [Mitchell & Plotkin 88] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. on Programming Languages and Systems*, 10(3), July 1988. (Published earlier in POPL 86.).
- [Sheldon 90] Mark A. Sheldon. *Static Dependent Types for First-Class Modules*. Technical Report MIT/LCS/TR-472, M.I.T. Laboratory for Computer Science, 1990.