Static Dependent Types for First-Class Modules

by

Mark A. Sheldon

B.S., Duke University (1984)

Submitted to the Department of
Electrical Engineering and Computer
Science in partial fulfillment of the
requirements for the degree of

Master of Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1989

Signature of Author______________________________
Department of Department of Electrical Engineering and Computer Science
22 May 1989

Certified by______________________________
David K. Gifford
Associate Professor of Computer Science
Thesis Supervisor

Accepted by______________________________
Arthur C. Smith
Chairman, Departmental Committee on Graduate Students

Static Dependent Types for First-Class Modules

by

Mark A. Sheldon

Submitted to the Department of
Department of Electrical Engineering and Computer Science
on 22 May 1989 in partial fulfillment of the
requirements for the degree of Master of Science in
Department of Electrical Engineering and Computer Science

Abstract

Static dependent types are the basis of a new type system which allows types and values to be packaged together in first-class modules, permitting flexible use of packaged types while retaining static decidability. Previous type systems restrict the use of modules, restrict access to packaged types, or fail to provide static type checking. The use of static *effect* information guarantees type soundness in the presence of side effects. Experience with an implementation of static dependent types in the *FX* programming language demonstrates their power. In particular, static dependent types can be used to implement types that are ordinarily built-in, and permit *FX* to be its own linking language.

Thesis Supervisor: David K. Gifford
Title: Associate Professor of Computer Science

Acknowledgments

I would like to express my gratitude to my advisor, Professor David K. Gifford, for his good advice, sound judgment, and infinite patience. His endless supply of ideas and encouragement have been a crucial ingredient in this work and have made the project both interesting and enjoyable.

I would also like to thank the members, past and present, of the Programming Systems Research Group and the Laboratory for Computer Science. In particular, Rebecca Bisbee, Michael Blair, Mark Day, Pierre Jouvelot, John Lucassen, James W. O'Toole Jr., Jonathan Rees, Mark B. Reinhold, David Andrew Segal, and Franklyn Turbak all made important contributions to this work. Jonathan Rees, Franklyn Turbak and David Segal made many useful comments on later drafts of this thesis.

For sustenance, I would like to thank my cooking club and the residents of Ashdown House. I thank Ishrat Chaudhuri for her loving support.

Last, but not least, I would like to express my gratitude to the MIT Ballroom Dance Club, my dance coaches, and my dance partner Yanina Kisler. Without their help, I would certainly have been less sane.

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

Contents

Chapter 1

Introduction

Static dependent types are the basis of a new type system which allows types and values to be packaged together in first-class modules, permitting flexible use of packaged types while retaining static decidability. Previous type systems restrict the use of modules, restrict access to packaged types, or fail to provide static type checking. The use of static *effect* information guarantees type soundness in the presence of side effects. Experience with an implementation of static dependent types in the *FX* programming language [Gifford, *et al.* 87] demonstrates their power. In particular, static dependent types can be used to implement types that are ordinarily built-in, and to permit *FX* to be its own linking language.

1.1 Motivation

Programmers tackle the complexity of a problem by decomposing it into smaller, more manageable subproblems. To support this process, programming languages usually provide some means to decompose programs into smaller, more easily understood, program modules. Programmers may develop and test modules independently and then combine them into a complete system. Moreover, useful modules may often be shared among several programs, thus saving the effort of recoding solutions to solved problems.

Modules usually provide tools of related functionality. For example, a set of matrix manipulation subroutines, *e.g.* for matrix inversion, transposition, gaussian elimination, might be grouped together in one module.

The values in a module may implement an abstract behavior like stacks, process queues, or hash tables. In a typed language, the programmer would like to define a new data type which the module is said to implement. To prevent users of the data type from violating internal invariants, the module may hide the definition of the abstract type and prevent access to values of that type except through the module. This allows module creation to be more independent of module use.

The components of a module usually have names to provide easy access to them. It is for this reason that modules are sometimes called *environments* [MacQueen 84].

Definition 1 *A* module *is a collection of related named data types and/or values.*[1]

1.2 Background

A module system design represents a choice in the trade-off between expressive power and complexity. Most previous systems restrict the use of modules, usually requiring all modules to be defined at top-level at compile time. The few systems which do allow first-class modules do not have side effects and lack either flexibility or decidability.

1.2.1 Most Module Systems are Second-Class

Languages like CLU [Liskov, *et al.* 81] and Ada [DoD 83] allow the programmer to package a data type and a set of operations together into a module which may be compiled separately and then used by other programs. However, they require that all references to modules be statically resolvable: module references must be manifest constants.

[1]This is not the most general definition. There is no reason why modules may not also contain macros, for example. However, a more general definition would only add confusion arising from issues not relevant to the current project.

Requiring module references to be manifest makes writing some programs more difficult. For example, there may be different modules in a system implementing the same behavior, or *abstraction*, optimized for different cases. For example, there may be two matrix implementations in a system, one for sparse and one for dense matrices. A programmer may need to use the matrix abstraction without knowing which implementation is more appropriate to the data, which will not be available until run time. The most straightforward approach is to write the code once, relying only on the existence of *some* matrix implementation, and choose the more efficient one dynamically. This code can be packaged in a subroutine:

```
P = (lambda (matrix-impl)
      ...create and manipulate matrices using the data...)
```

Then, the programmer can check the data and apply `P` to the appropriate module dynamically.

```
(P (if (sparse? data) sparce-matrix-impl dense-matrix-impl))
```

In most systems, modules may not be passed to subroutines as arguments. The result is that the code in the subroutine is duplicated for each case.

Definition 2 *A value is* first-class *if it can be named, passed to subroutines, returned from subroutines, and stored in data structures.*

Most programming languages do not support first-class modules.

Second-class modules have certain implementation advantages. Ada, for example, uses the presence of static representation information to optimize runtime representations of values, *i.e.,* it *unboxes* them. However, this necessitates recompiling all users of a module when the module changes — a violation of abstraction principles. Second-class modules also avoid the problem of maintaining type safety in the presence of side effects: it is easy for the implementation to guarantee that there is no way to mutate a module implementation.

Unfortunately, second-class modules increase the intellectual overhead of programming because they require a separate linking language to combine modules together. All the tools the programmer uses to write programs, *e.g.* subroutine abstraction, are unavailable for structuring large systems.[2] ML [MacQueen 84] makes an effort to ameliorate this problem by making the linking language mesh fairly well with the rest of the language, but it still has a separate linking language with separate rules governing its use.

1.2.2 Weak Existential Types are Inflexible

Research in the area of data abstraction has benefitted from work in logic. In particular, investigators have imported the notions of type *quantification* from [Girard 72] and [Martin-Löf 73].

The SOL programming language [Mitchell and Plotkin 88] uses the notion of existential quantification for abstract data types and allows values of existential type to be first-class.

Definition 3 *An* existential type *packages types and values together into a single module value. The type names are bound in an existential type and are accessible in the types of the component values.*

For example, a module that implements stacks of integers would have the type:

```
(∃ (intstack) ((mkstack :  () → intstack)
               (push    :  intstack × int → intstack)
               (pop     :  intstack → int × intstack)))
```

Suppose `intstackimpl` is a module with the above type. In SOL, the programmer may use the module in an `abstype` construct:

[2][Burstall and Lampson 84] makes the observation that first-class modules allow uniform treatment of ordinary computation and module linking.

```
abstype intstack with mkstack :  () → intstack,
                      push    :  intstack × int → intstack
                      pop     :  intstack → int × intstack
   is intstackimpl
   in
...code which uses stacks...
end
```

SOL does not allow the bound type variable of an `abstype` to appear free in the type of the `abstype` return value. Consequently, there is no way to refer to the type implemented by a module outside of an `abstype`.[3]

But, one might like to return a stack from such an expression. Or, since modules are first-class, one may want to operate on any stack regardless of which implementation created it. Consider the subroutine for reversing all the elements of a stack:

```
reverse-stack = (lambda (stack-impl a-stack)
                  ;; Make a new stack.  Pop the elements
                  ;; off the old stack and push them onto
                  ;; the new one.  Return the new stack.
                 )
```

SOL does not provide any way to write the type of such a subroutine because the type of `a-stack` and the return type depend on `stack-impl`.

Definition 4 *A module type is* generally available *if it can be extracted from a module anywhere in the program.*[4]

Definition 5 *An existential type system is* weak *if its module types are not generally available.*

[3]SOL also does not allow the bound type variable of an `abstype` to appear free in the types of any free variables in the `abstype` body. This restriction, however, has nothing to do with the flexibility of the module system: it is intended to prevent capture. Alpha-renaming the program would eliminate the need for this restriction. See Section 3.1 for a discussion of this issue.

[4]The author is grateful to Jonathan Rees for suggesting the name *generally available*.

Languages with second-class module systems, like CLU and Ada, have an advantage in this regard: They force the programmer to name all modules and provide their definitions at top-level. This way, all the types implemented by modules used by a program are available everywhere in the program.

1.2.3 Strong Existential Types are Undecidable

To allow module types to be generally available, type expressions must be able to refer to values.

Definition 6 *A* dependent type *is one containing a value expression.*

The Pebble programming language [Burstall and Lampson 84] provides this facility by allowing types to be first-class values. This means that the programmer may refer to a type in a module by selecting it out as if it were a value.

Definition 7 *An existential type system is* strong *if it allows types to be treated as first-class values.*[5]

Because a type can be the result of arbitrary computation in a strong existential type system, type checking may fail to terminate. In addition, strong existential types are not truly abstract because the program enclosing the module definition can select the representation type out of a module explicitly. This is why Pebble uses a password mechanism in implementations of abstract types. The implementation can check to see that values were not manufactured elsewhere by checking the password. (See [Burstall and Lampson 84, pp. 18–19].) This mechanism is *ad hoc*, and, in light of the current proposal, unnecessary.

[5]Usually, the definition of strong existential type states that the carrier of a module is a full-fledged type. The definition given here makes use of the result of [Hook and Howe 86] which shows that having strong existential types, in this sense, is equivalent to having the `type : type` axiom. [Meyer and Reinhold 86] shows that the `type : type` axiom implies that the type system is undecidable.

1.3 Goals

The challenge is to design a language which simultaneously provides:

- Type safety.
- Data abstraction.
- Multiple (recursive) abstractions.
- Higher order abstractions.
- First-class module implementations.
- Safe interaction with side effects.
- General availability of module types.
- Static type checking.

Many languages meet some of these goals: ML has multiple, recursive abstract types,[6] simple type constructors, side effects, and static type checking. SOL does not have side effects but allows first-class modules. Pebble meets all the goals except static type checking (though data abstraction relies on an *ad hoc* password scheme). Quest [Cardelli 89] has a complete system of higher order abstractions but does not have dependent subroutines and does not provide general availability.

The following table presents a survey comparing some current languages with our system of static dependent types (SDT) according to these goals. (All the given languages provide type safety and data abstraction.)

	CLU	Ada	ML	SOL	Quest	Pebble	SDT
Mult. ADTs			●		●	●	●
Higher Order					●	●	●
1st Class Mods				●	●	●	●
Side Effects	●	●	●		●		●
Gen. Avail.						●	●
Stat. Check.	●	●	●	●	●		●

[6]An implementation of trees which exports abstractions for both `node` and `tree` is an example of the use of multiple recursive types.

1.4 The Proposal

1.4.1 Static Dependent Types

Definition 8 *A* static dependent type (SDT) *system provides general availability of module types and preserves the separation of types and values.*

To preserve static type checking, a static dependent type system will not allow types to be the results of arbitrary computation and will not require the full evaluation of values in dependent types. Instead, there will be a single type which may directly contain a value expression. The expression in a dependent type will *not* be evaluated; it will be statically compared to expressions in other dependent types (in this case using simple textual equality), and it will have variables replaced by their definitions whenever the dependent type is exported out of the scope of some of its free variables.

Quest [Cardelli 89] attempted the same sort of trade-off, but, because Quest has no way of restricting side effects, it restricts dependent types so that only variables may appear in them. This means that the substitutions which allow dependent types to be meaningful when exported out of a scope cannot take place.

1.4.2 *FX* is a Good Base Language

An *effect system* like the one in *FX* [Gifford, *et al.* 87], based on [Lucassen 87], provides a means of specifying and enforcing constraints which guarantee type safety in the presence of first-class modules, dependent types, and side effects. The design of the static dependent type system will contain effect constraints enforced by the effect system.

Also, the *FX kind system* based on [McCracken 79] supports the notion of a *description*. A description is a type, an effect, a region of the store, or a function from descriptions to descriptions. Thus, *FX* provides higher order descriptions. In fact, *FX* provides all the power of the second order typed lambda calculus.

1.5 Structure of the Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 gives the design of a simple system built on top of the *FX-87* kernel of Appendix A. The design is specified as a grammar, static semantics, and informal dynamic semantics with examples.
- Chapter 3 describes an implementation built into the *FX-87* Interpreter [Jouvelot and Gifford 88] to test the utility and practicality of the module system design.
- Chapter 4 summarizes the results of the design and implementation efforts and describes directions for current and future research.

Chapter 2

Static Dependent Types

The *FX-87* kernel described in Appendix A may be extended to support first-class modules and static dependent types (SDTs). The syntax description given here shows only new or changed forms; other forms remain as given in the appendix. Similarly, the static semantics provides only the new and changed kinding and typing rules.

For readability, examples will often leave out projections of polymorphic values onto type arguments. In all such cases, an implicit projection mechanism like the one in *FX-87* could supply the missing arguments automatically. (In fact, the implementation described in Chapter 3 retains this mechanism.)

The design presented here is one in a series of static dependent type systems. The kernel idea, that there should be a static semantics for expressions in dependent types is due to Mark Reinhold. He worked out a scheme in a language without side effects and without higher order descriptions. The author generalized and implemented this system with higher order descriptions, effects, and regions. Then, the author, with Reinhold, extended the design to include multiple, mutually recursive abstractions and multiple, mutually recursive values. The author implemented this system as well. We have also created a design with transparent description bindings (descriptions whose representations are deliberately exposed outside the module) for a new version of *FX*. This extended design, however, has not yet been implemented.

2.1 Syntax

2.1.1 Descriptions

If modules are first-class values, then they must have types. The `modof` type gives the names and kinds of the abstractions exported by a module as well as the names and types of the values exported by a module. The types in a `modof` type are in the scope of the abstraction names. Dependent subroutine types bind the names of the formal parameters of the subroutine. Subsequent parameter types and the return type may refer to the preceding formal names.

Texp = – Types

. . .
(`modof` ((*id Kexp*). . .) ((*id Texp*). . .))
(`subr` *Eexp* ((*id Texp*). . .) *Texp*)

Dependent types (possibly higher order) arise from the presence of the `dselect` form. The expression, *Exp*, in a `dselect` must denote a module. The name, *id*, specifies an abstraction name exported by the module.

GDesc = – Generic descriptions

. . .
(`dselect` *Exp id*)

2.1.2 Expressions

The new expressions allow one to construct modules (`mod`), to access the values and descriptions exported by a module (`with`), and to read values from a file which must exist at compile time (`input`). The `input` expression is the key to linking separately written modules.

Exp = – Expressions

...
(`mod` ((*id Kexp Dexp*)...) ((*id Texp Exp*)...))
(`with` *Exp Exp*)
(`input` *string-literal*)

2.1.3 Syntactic Sugar

Dotted identifiers provide a useful shorthand for referring to abstractions or values in a module.

Dotted-id = – Dotted Identifiers

id | *Dotted-id.id*

A dotted identifier which is just an identifier desugars to itself.

If a dotted identifier appears where a description is expected, then the following desugaring takes place:

Dotted-id.id ⇒ (`dselect` *Dotted-id id*)

If a dotted identifier appears where an expression is expected, then the following desugaring takes place:

Dotted-id.id ⇒ (`with` *Dotted-id id*)

Overloading the dot notation is not a problem: the parser always knows whether it expects a description or an expression. Dotted identifiers are expanded accordingly.

The formal names in subroutine types are useful both for dependent subroutines and for documentation. However, sometimes the name of a formal is really unimportant. For this reason, the old subroutine type syntax of Appendix A may still be accepted. Such old subroutine types are equivalent to new ones with automatically generated formal names. (The chosen formal names must not capture any free variables in the formal and return types.)

2.2 Static Semantics

Each inference rule is given in a separate section together with motivation and explanation. Each description ends with one or more programming examples.

2.2.1 Kind Inference Rules

Modof

A `modof` description, *i.e.*, a module interface, is a type. Abstractions may only be types or (possibly higher order) type constructors. The rational for this restriction appears with the `mod` typing rule on page 26.

$$\frac{\begin{array}{c}\mathrm{TK}[^{n}_{i=1} ida_i \;\texttt{::}\; \kappa_i] \vdash \tau_j \;\texttt{::}\; \texttt{type} \quad (1 \le j \le m) \\ \mathrm{FinallyType}(\kappa_i) \quad (1 \le i \le n)\end{array}}{\begin{array}{l}\mathrm{TK} \vdash (\texttt{modof}\ ((ida_1\ \kappa_1)\ldots(ida_n\ \kappa_n)) \\ \qquad\qquad ((idv_1\ \tau_1)\ldots(idv_m\ \tau_m)))\ \texttt{::}\ \texttt{type}\end{array}}$$

FinallyType has the recursive definition:

$$\begin{array}{rcl}\mathrm{FinallyType}\,(\texttt{type}) & = & \mathit{true} \\ \mathrm{FinallyType}\,(\ (\texttt{dfunc}\ (\kappa_1\ldots)\ \kappa)\ ) & = & \mathrm{FinallyType}\,(\kappa)\end{array}$$

For example, `pair-type` might be the interface of a module implementing mutable pairs:

`pair-type` $\equiv$

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

A module of this type must define a type constructor (a description function mapping two types and a region to a type), as well as polymorphic subroutines for pair creation, destructuring, and mutation.

Dselect

The `dselect` form is an example of what is often called a *witness* operator: it allows the programmer to refer to an abstraction implemented by a module. Because a `dselect` description contains an expression, it is a *dependent description.*

$$\frac{\begin{array}{c}\mathrm{TK} \vdash e : (\texttt{modof}\ ((\mathit{ida}_1\ \kappa_1)\ldots)\\ \qquad\qquad\qquad\qquad ((\mathit{idv}_1\ \tau_1)\ldots))\\ !\ \epsilon\\ \epsilon \sqsubseteq (\texttt{maxeff}\ (\texttt{write}\ \rho_1)\ldots(\texttt{alloc}\ \rho_1)\ldots)\quad \forall\rho_i\end{array}}{\mathrm{TK} \vdash (\texttt{dselect}\ e\ \mathit{ida}_i) :: \kappa_i}$$

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. An earlier proposal of this system required that the module expression in a `dselect` be `pure`. But this is too restrictive. If the module expression only allocates and writes, there is no problem. Only if it reads mutable storage which another expression may alter is it possible for an implementation to change dynamically.[1] This looser restriction is enforced in the preceding rule by the requirement that the effect of the module expression consist of zero or more `write` and `alloc` effects.

Design Constraint 1 *Value expressions occurring in descriptions must not have* `read` *effects.*

The ability to articulate and enforce this effect restriction allows this system to do what other languages do not: to combine first-class modules, dependent types, and 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 comparisons to other languages in Chapter 1.)

The kernel language of Appendix A has the property that, though the type-checking rules often require kind deductions, the kind-checking rules never require type deductions. The `dselect` kinding rule violates this principle. Will some programs create infinite chains of deductions? Expressions written by programmers are always finite and cannot contain themselves. As long as type-checking the expression in a `dselect` 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 ordinary variables (in TK) are needed. Provided that these types are already known and kind-checked, the deduction will be finite. Given the binding constructs of the kernel language, this is certainly true. The design of each binding construct introduced here respects this principle:[2]

[1]Pierre Jouvelot first made this observation.

[2]The restrictions implied by this scheme may be overly conservative, but proofs of

Design Constraint 2 *Free ordinary variables in a description must already have their types kind-checked and in TK. Equivalently, whenever an ordinary variable is introduced, its type may not depend directly or indirectly on the that variable.*

If `pair-impl` were a variable bound to a module with the above `modof` type, then one may use the `dselect` form to extract the abstraction:

```
(dselect pair-impl pairof) :: (dfunc type type region)
```

Calling `mk-pair` to create a pair of integers in the region `@foo` would produce a result of type:

```
((dselect pair-impl pairof) int int @foo)
```

One may abbreviate this type using dotted identifier notation:

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

Subr

Dependent subroutine types are binding constructs. Each formal name is available to the types of all later formals and to the return type. No formal name is available to its own type, however.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \epsilon & :: & \texttt{effect} \\ \mathrm{TK}[_{i=1}^{j} id_i : \tau_i] \vdash \tau_{j+1} & :: & \texttt{type} \quad (0 \leq j \leq n) \end{array}}{\mathrm{TK} \vdash (\texttt{subr}\ \epsilon\ ((id_1\ \tau_1) \ldots (id_n\ \tau_n))\ \tau_{n+1})\ :: \texttt{type}}$$

termination become much more complicated if the restrictions are relaxed. I have not been able to write any programs which make use of more relaxed rules: The programs that exceed the limitations always seem to be untypable.

Dependent subroutines are useful for writing programs that take both an implementation of an abstraction and an object of the abstract type:

```
(subr (read @foo) ((m pair-type)
                   (p ((dselect m pairof) int int @foo)))
  ((dselect m pairof) int char @foo))
```

Because of the parenthesized syntax of *FX*, such types are sometimes difficult to read. Using a more mathematical notation, this subroutine type might be written:

```
m:pair-type × p:(m.pairof int int @foo)
                  ⟶
            (read @foo) (m.pairof int char @foo)
```

Dependent subroutines are also useful for doing module linking. Chapter 3 provides a practical example of this.

2.2.2 Type and Effect Inference Rules

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. No formal name is available to its own type, however. Since abstractions are restricted to be (possibly higher order) type constructors, it does not matter whether or not the latent effect of the subroutine is in the scope of the formals. There is no way to get a region or effect out of a `dselect`.

$$\frac{\begin{array}{rcl} \mathrm{TK}[^{j}_{i=1} id_i : \tau_i] \vdash \tau_{j+1} & :: & \texttt{type} \quad (0 \leq j \leq n) \\ \mathrm{TK}[^{n}_{i=1} id_i : \tau_i] \vdash e & : & \tau_{n+1} \; ! \; \epsilon \end{array}}{\begin{array}{ll} \mathrm{TK} \vdash & (\texttt{lambda}\ ((id_1\ \tau_1) \ldots (id_n\ \tau_n))\ e) \\ & \quad : (\texttt{subr}\ \epsilon\ ((id_1\ \tau_1) \ldots (id_n\ \tau_n))\ \tau_{n+1}) \\ & \quad !\ \texttt{pure} \end{array}}$$

The subroutine `pair-example` is a dependent subroutine:

`pair-example` $\equiv$

```
(lambda ((m pair-type) (p (m.pairof int int @foo)))
  ((proj m.mk-pair @foo) (m.fst p) (int->char (m.snd p))))
```

This subroutine takes a pair implementation and a pair created by that implementation. It returns a new pair consisting of the same first element as its argument pair and the character equivalent of the integer in the second element of the argument pair.

Application

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

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

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. If one does not, then the resulting expression will not be well-kinded.

Suppose `pair-impl` has type `pair-type`. Then the application:

```
(pair-example pair-impl
              ((proj pair-impl.mk-pair @foo) 3 0))
```

has type `(pair-impl.pairof int char @foo)`. Notice that `pair-impl` was substituted for the formal parameter `m` in the type of `pair-example`.

Let

The `let` rule must change in the same way as the application rule. The bindings of a `let` expression are *opaque*: Even if two identifiers are bound to textually identical expressions, `dselect` descriptions from the two variables are *not* interconvertible. This is a consequence of making the semantics of `let` follow the semantics of `lambda`. One advantage of this arrangement is that it 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; *i.e.* the system enforces the same constraints as weak existential types in the presence of `read`s.

In accordance with Design Constraint 2 the types of the expressions bound by the `let` expression cannot depend on the names it binds.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e_i & \mathbf{:} & \tau_i \ \mathtt{!} \ \epsilon_i \quad (1 \leq i \leq n) \\ \mathrm{TK}[^n_{i=1} id_i \ \mathbf{:} \ \tau_i] \vdash e & \mathbf{:} & \tau \ \mathtt{!} \ \epsilon \\ \mathrm{TK} \vdash [^n_{i=1} e_i / id_i]\tau & \mathbf{::} & \mathtt{type} \end{array}}{\mathrm{TK} \vdash (\mathtt{let}\ ((id_1\ e_1) \ldots)\ e) \ \mathbf{:} \ [^n_{i=1} e_i / id_i]\tau \ \mathtt{!} \ (\mathtt{maxeff}\ \epsilon\ \epsilon_1 \ldots)}$$

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.

Suppose that `mk-pair-impl` is a subroutine which takes a string describing which of several possible pair implementations the caller desires. Then the following `let` expression chooses one called `"lambda-based"` and uses it to call the `pair-example` subroutine defined above.

```
(let ((pair-impl (mk-pair-impl "lambda-based")))
  (pair-example pair-impl
                ((proj pair-impl.mk-pair @foo) 0 0)))
```

The type of the above expression is:

```
((dselect (mk-pair-impl "lambda-based") pairof) int char @foo)
```

Mod

To build a module, one specifies a set of description bindings and a set of value bindings:

- A *description binding* is an identifier (the abstraction name), a kind, and a description which is the abstraction's representation. The abstraction definitions are mutually recursive.
- A *value binding* is an identifier (the field name), a type, and an expression which is the field definition. Field definitions are also mutually recursive. In accordance with Design Constraint 2, the types of the values are *not* in the scope of the field names.[3]

$$
\frac{\begin{array}{rcl}
\mathrm{TK}_1 & = & \mathrm{TK}[^n_{i=1} \mathit{ida}_i \; \texttt{::} \; \kappa_i] \\
\mathrm{TK}_1 \vdash \delta_i & \texttt{::} & \kappa_i \quad (1 \le i \le n) \\
\mathrm{Immutable}(\rho) & & \forall \rho \in \mathrm{FRC}(\delta_i) \quad (1 \le i \le n) \\
\mathrm{FinallyType}\,(\kappa_i) & & \quad (1 \le i \le n) \\
\mathrm{TK}_1 \vdash \tau_i & \texttt{::} & \texttt{type} \quad (1 \le i \le m) \\
\mathrm{TK}_2 & = & \mathrm{TK}_1[^n_{i=1} \texttt{up}-\mathit{ida}_i \; \texttt{:} \; \mathrm{UpType}(\mathit{ida}_i, \kappa_i, \delta_i)] \\
\mathrm{TK}_3 & = & \mathrm{TK}_2[^n_{i=1} \texttt{down}-\mathit{ida}_i \; \texttt{:} \; \mathrm{DownType}(\mathit{ida}_i, \kappa_i, \delta_i)] \\
\mathrm{TK}_3[^m_{i=1} \mathit{idv}_i \; \texttt{:} \; \tau_i] \vdash e_j & \texttt{:} & \tau_j \quad (1 \le j \le m)
\end{array}}{\begin{array}{rl}
\mathrm{TK} \vdash & (\texttt{mod}\ ((\mathit{ida}_1\ \kappa_1\ \delta_1) \ldots (\mathit{ida}_n\ \kappa_n\ \delta_n)) \\
& \quad ((\mathit{idv}_1\ \tau_1\ e_1) \ldots (\mathit{idv}_m\ \tau_m\ e_m))) \\
& \texttt{:}\ (\texttt{modof}\ ((\mathit{ida}_1\ \kappa_1) \ldots (\mathit{ida}_n\ \kappa_n)) \\
& \qquad ((\mathit{idv}_1\ \tau_1) \ldots (\mathit{idv}_m\ \tau_m))) \\
& \texttt{!}\ (\texttt{maxeff}\ \epsilon_1 \ldots \epsilon_m)
\end{array}}
$$

The definitions of UpType and DownType are given below.

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

[3]An implementation of a predecessor of this system failed to make this scope restriction and could indeed be made to loop forever.

The use of TK_1 in kind checking the representations allows the representations to be mutually recursive. This recursion is different from the recursion in the `dletrec` or `pletrec` forms of *FX-87*: there, recursion produced conceptually infinite descriptions (though they can be finitely represented). 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.

The Immutable predicate is defined in Appendix A. In *FX*, the only immutable region is `@=`. FRC is a function which computes the free region constants of a description in the obvious way. Forbidding free mutable region constants in representations forces programmers to parameterize their mutable abstractions over any regions. This restriction is not intended merely to enforce a particular programming style. Embedding mutable regions in representations would hide the passage of mutable values through `up` and `down` calls and would allow side effects on those values to be masked.[4] This could cause scheduling errors in parallel implementations [Hammel and Gifford 88] or cause optimizations like memoization and stack allocation of short-lived objects to be applied unsafely in sequential implementations [Lucassen 87].

Design Constraint 3 *Abstraction boundaries must not hide side effects.*

The following module, if allowed, would turn the usual reference type, `refof`, with operations `new`, `get`, and `set` into an effect loophole. The subroutines all have latent effect `pure` because the region `@foo` does not appear in the types of the free variables of the `lambda` bodies, nor does it appear in the return types.

[4]For a definition of effect masking see page 57 of Appendix A.

```
(mod ((t (refof int @foo)))
     ((anew (subr pure ((x int)) t)
            (lambda ((x int))
              (up-t ((proj new @foo) x))))
      (aget (subr pure ((x t)) int)
            (lambda ((x t))
              ((proj get @foo) (down-t x))))
      (aset (subr pure ((x t) (y int)) unit)
            (lambda ((x t) (y int))
              ((proj set @foo) x y)))))
```

An alternative to this restriction would be to eliminate the effect masking rule. But this would eliminate opportunities for optimization. Disabling the effect masking rule just within the `mod` typing rule would be non-uniform and would still inhibit optimizations. (Recall that the `mod` expression is the source of all recursion in the language.)

Abstract effect constructors could be useful for specifying that operations defined in some module do not interfere with *any* outside computations, regardless of accidental choice of region names. But if one can use this power to change the apparent latent effect of subroutines, then an effect loophole opens up. Determining how to add abstract effects in a safe and useful way is the subject of current research.[5]

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. Perhaps the simplest method in a system with a fully specified module signature is to allow abstract names to be interconvertible with their representations throughout the module definition. The resulting `modof` type is then taken from the signature. This approach is taken by Quest [Cardelli 89]. This system has two failings:

[5]The author and Pierre Jouvelot have investigated this issue and examined possible applications.

- The module writer cannot use the abstractions locally to enforce invariants. This is especially important for large modules which define multiple abstractions.
- The `modof` type will be underspecified if the explicit interface is omitted when type reconstruction is added to the language.[6]

Another approach is to provide a form where the programmer specifies what names are interconvertible with their representations within the form and what the result type should be. This is similar to the above idea, where conversion happens everywhere, but the programmer can control the scope and extent of the conversion. Such a form is very general, but unnecessarily complicated when the usual case is very simple.

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.) Implicit projection often allows the programmer to ignore the fact that these subroutines are polymorphic. 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.

$$
\begin{array}{l}
\text{UpType}(\delta_a, \texttt{type}, \delta_r) = \\
\quad (\texttt{subr pure } ((x\ \delta from))\ \delta_a) \\
\text{UpType}(\delta_a, (\texttt{dfunc } (\kappa_1 \ldots \kappa_n)\ \kappa_{n+1}), \delta_r) = \\
\quad (\texttt{poly } ((x_1\ \kappa_1) \ldots (x_n\ \kappa_n)) \\
\qquad \text{UpType}(\ (\delta_a\ x_1 \ldots x_n), \kappa_{n+1}, (\delta_r\ x_1 \ldots x_n)\ )) \\
\text{DownType}(\delta_a, \kappa, \delta_r) = \\
\quad \text{UpType}(\delta_r, \kappa, \delta_a)
\end{array}
$$

An implementation of complex numbers based on pairs and floating point numbers is a simple and useful example of a module. If `ipairof` represents

[6]Type reconstruction issues are beyond the scope of this thesis, but it is a design goal of *FX* to allow as many declarations as possible to be omitted. For current results of the FX type reconstruction effort, see [O'Toole and Gifford 89, O'Toole 89].

the type constructor of immutable pairs, and `icons`, `icar`, and `icdr` the corresponding constructor and destructuring operations, then the following module implements complex numbers:

```
(mod ((complex (ipairof float float)))
     ((origin (up-complex (icons 0.0 0.0)))
      (make-complex (lambda ((x float) (y float))
                      (up-complex (icons x y))))
      (get-x (lambda ((c complex))
               (icar (down-complex c))))
      (get-y (lambda ((c complex))
               (icdr (down-complex c))))
      (get-rho (lambda ((c complex))
                 (let ((c (down-complex c)))
                   (sqrt
                     (fl+ (fl* (icar c) (icar c))
                          (fl* (icdr c) (icdr c)))))))
      (get-theta (lambda ((c complex))
                   (let ((c (down-complex c)))
                     (atan (fl/ (icar c) (icdr c))))))
      (= (lambda ((c1 complex) (c2 complex))
           (let ((c1 (down-complex c1))
                 (c2 (down-complex c2)))
             (and (fl= (icar c1) (icar c2))
                  (fl= (icdr c1) (icdr c2))))))
      (+ (lambda ((c1 complex) (c2 complex))
           (let ((c1 (down-complex c1))
                 (c2 (down-complex c2)))
             (up-complex
               (icons (fl+ (icar c1) (icar c2))
                      (fl+ (icdr c1) (icdr c2)))))))
      (- (lambda ((c1 complex) (c2 complex))
           (let ((c1 (down-complex c1))
                 (c2 (down-complex c2)))
             (up-complex
               (icons (fl- (icar c1) (icar c2))
                      (fl- (icdr c1) (icdr c2))))))))
```

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.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e_m & : & (\texttt{modof}\ ((ida_1\ \kappa_1)\ldots(ida_n\ \kappa_n)) \\ & & \qquad ((idv_1\ \tau_1)\ldots(idv_m\ \tau_m))) \\ & \texttt{!} & \epsilon_m \\ \mathrm{TK}[ida_1 :: \kappa_1\ldots][idv_1 : \tau_1\ldots] \vdash e & : & \tau\ \texttt{!}\ \epsilon \\ \tau' & = & [^m_{j=1}(\texttt{with}\ e_m\ idv_j)/idv_j]\tau \\ \tau'' & = & [^n_{i=1}(\texttt{dselect}\ e_m\ ida_i)/ida_i]\tau' \\ \mathrm{TK} \vdash \tau'' & :: & \texttt{type} \end{array}}{\mathrm{TK} \vdash (\texttt{with}\ e_m\ e) \ : \ \tau''\ \texttt{!}\ (\texttt{maxeff}\ \epsilon_m\ \epsilon)}$$

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 `dselect`s, and field names are replaced by `with` expressions.

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.

Inside a `with` expression, the programmer may refer directly to the names bound by a module. The `with` expression:

```
(with pair-impl
  (the (pairof int bool @bar) ((proj mk-pair @bar) 0 #t)))
```

has type `(pair-impl.pairof int bool @bar)`.

Input

The `input` form allows programs to be split into multiple files. The file named in an `input` form *must* exist at compile time. The only free variables allowed in such a separate file are those defined by the *FX* library. (See

Section 3.6 for a description of the `fx` module.) ϕ represents the empty type and kind environment.

$$\frac{\phi \vdash (\texttt{with fx}\ \mathrm{FS}(\textit{string-literal}))\ \texttt{:}\ \tau\ \texttt{!}\ \epsilon}{\mathrm{TK} \vdash (\texttt{input}\ \textit{string-literal})\ \texttt{:}\ \tau\ \texttt{!}\ \epsilon}$$

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 demonstrates how this can be implemented on a standard file system. (See page 40.)

The program

```
(let ((pair-impl (input "PSRG:FX;IMPL;PAIROF.FX")))
  (with pair-impl
    ((proj mk-pair @baz) #t #f)))
```

has type

```
((dselect (input "PSRG:FX;IMPL;PAIROF.FX") pairof) bool bool @baz)
```

See Section 3.6 for examples of linking using file input and subroutines.

2.3 Inclusion Rules

The following inclusion rules are either additions or changes to the inclusion relation $\sqsubseteq$ defined in Appendix A. The rules can be thought of as defining a reduction semantics for descriptions with α, β, and η conversion. Because this language does not have the transparent recursive types of *FX-87*, the kinded system of descriptions corresponds to the simply typed lambda calculus [Berendregt 84], and all descriptions have normal forms modulo alpha-renaming.

The new subroutine type admits alpha-renaming:

$$\frac{\begin{array}{c} id_i{}' \notin \mathrm{FV}(\tau_j) \quad (1 \leq i \leq n) \quad (2 \leq j \leq n+1) \\ \tau_j' = [_{i=1}^{n} id_i{}'/id_i]\tau_j \quad (2 \leq j \leq n+1) \end{array}}{(\texttt{subr}\ \epsilon\ ((id_1\ \tau_1) \ldots (id_n\ \tau_n))\ \tau_{n+1}) \equiv (\texttt{subr}\ \epsilon\ ((id_1{}'\ \tau_1') \ldots (id_n{}'\ \tau_n'))\ \tau_{n+1}')}$$

The inclusion rule on subroutine types is the same as the old rule, though it may be used in conjunction with the above alpha-renaming rule.

$$\frac{\begin{array}{rcl} \epsilon & \sqsubseteq & \epsilon' \\ \tau_i' & \sqsubseteq & \tau_i \quad (1 \leq i \leq n) \\ \tau & \sqsubseteq & \tau' \end{array}}{(\texttt{subr}\ \epsilon\ ((id_1\ \tau_1) \ldots (id_n\ \tau_n))\ \tau) \sqsubseteq (\texttt{subr}\ \epsilon'\ ((id_1\ \tau_1') \ldots (id_n\ \tau_n'))\ \tau')}$$

One may freely permute the fields in a `modof` type:

$$\frac{\begin{array}{c} \pi \text{ is a permutation on } [1,n] \\ \pi' \text{ is a permutation on } [1,m] \end{array}}{\begin{array}{c} (\texttt{modof}\ ((ida_1\ \kappa_1) \ldots (ida_n\ \kappa_n)) \\ \quad ((idv_1\ \tau_1) \ldots (idv_m\ \tau_n))) \\ \sqsubseteq \\ (\texttt{modof}\ ((ida_{\pi(1)}\ \kappa_{\pi(1)}) \ldots (ida_{\pi(n)}\ \kappa_{\pi(n)})) \\ \quad ((idv_{\pi'(1)}\ \tau_{\pi'(1)}) \ldots (idv_{\pi'(m)}\ \tau_{\pi'(m)}))) \end{array}}$$

To get a subtype of a `modof` type, one may add new fields and/or take subtypes of the fields.

$$\frac{\tau_i' \sqsubseteq \tau_i \quad (1 \leq i \leq n)}{\begin{array}{c} (\texttt{modof}\ ((ida_1\ \kappa_1) \ldots (ida_n\ \kappa_n) \ldots)((idv_1\ \tau_1') \ldots (idv_m\ \tau_n') \ldots)) \\ \sqsubseteq \\ (\texttt{modof}\ ((ida_1\ \kappa_1) \ldots (ida_n\ \kappa_n))((idv_1\ \tau_1) \ldots (idv_m\ \tau_n))) \end{array}}$$

There is no proper inclusion on `dselect` forms, only equivalence.

$$\frac{e \equiv e'}{(\texttt{dselect}\ e\ id) \equiv (\texttt{dselect}\ e'\ id)}$$

The `dselect` equivalence rule requires that the $\equiv$ relation be defined on value expressions (elements of *Exp*). There are many choices for the definition: 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 α and η conversion, and limited β 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 for present purposes.[7]

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

[7]A formal definition of textual equality is straight-forward and tedious. It is therefore omitted.

Chapter 3

Implementation

Implementation experience shows that the SDT system described in Chapter 2 can be implemented and that it is powerful enough to allow *FX* to be its own linking language.

In addition, the implementation has provided, and continues to provide, valuable feedback into the design process. Implementing and using SDTs clarifies the power, benefits, interactions, and limitations of language features. For example, experimentation exposed weaknesses in predecessors to the `up-` and `down-` subroutines described on page 28. Implementation problems exposed errors in the typing rules: the restrictions on free region constants in representations and the availability of module field names in field types are prominent examples.

The *FX-87* Interpreter described in [Jouvelot and Gifford 88] was the starting point of this implementation. Modifying the implementation to support the SDT system of Chapter 2 was quite straightforward. Kind, type, and effect checking for new and modified forms is, for the most part, a direct implementation of the rules from the static semantics. For example, the code in the implementation which computes the kind of a `modof` form is:

```
(define (kind-of-modof node tk-env dstore)
  (let ((abstractions (modof-abstractions node))
        (values (modof-values node)))
    (and
      (tst? (every? (lambda (bind) (kexp? (cadr bind)))
                    abstractions)
            "Cannot kind-check abstraction bindings in MODOF")
      (tst? (every? (lambda (bind) (finally-type? (cadr bind)))
                    abstractions)
           "Abstraction must be type or type constructor in MODOF")
      (tst? (every?
              (lambda (bind)
                (texp? (cadr bind)
                       (add-tk-env tk-env abstractions *kind*)
                       (add-dstore dstore
                                   (map (lambda (bind)
                                          `(,(car bind)
                                            (,(make-d-variable
                                                (car bind))
                                             ,(make-d-variable
                                                (car bind)))))
                                        abstractions))))
              values)
            "Invalid type expression in MODOF")
      (make-kind-type))))
```

The code checks that the abstraction bindings contain valid kinds which may be types or higher order type constructors. Then it checks that the types of the fields are valid types when the abstractions bindings are in the `tk-env` (the analog of TK). If all goes well, then the kind of the `modof` form is `type`. (The purpose of the `dstore` is explained below.)

In addition to the modifications to the interpreter, the implementation packages all the built-in types, type constructors, and subroutines into an `fx` module. All user programs may assume that they are surrounded by `(with fx ...)`.

3.1 Alpha-Renaming

In the `plambda` rule of [Lucassen 87, p. 43] and of [Gifford, *et al.* 87, p. 129], the bound description variables may not capture the free description variables in the types of the free ordinary variables of the body. This restriction naturally applies to `plet` expressions as well. The advent of dependent descriptions requires that the restriction also be observed in `dlambda` and `dlet` expressions. However, with the assumption that there are no duplicate bindings in the type and kind environment (TK),[1] the inference rules do not need to enforce this restriction because the undesirable situation can never arise.

Within any invocation of the *FX* Interpreter, all programs are alpha-renamed. This guarantees that all programs observe the restriction on the type and kind environment by eliminating all duplicate bindings.

Alpha-renaming a variable consists of appending an integer to the end of its name. So that alpha-renamed variables are readily distinguishable (for debugging purposes), the `@` character is placed between the name and the number. (The `@` character may not appear in variable names in legal input programs.)

The implementation creates a global environment for alpha-renaming when it is loaded. This `alpha-env` has two components: an environment mapping identifiers from the input program to the alpha-renamed identifiers which replace them (the `pending` environment) and a structure mapping identifiers to the number of times they have been introduced since system invocation (`cur-nums`). The `pending` environment consists of frames built up in accordance with the lexical structure of the program. When variables are introduced at the beginning of some lexical scope, the parser adds a new frame to the front of the `pending` environment mapping the identifiers to names computed from the information in `cur-nums`. `Cur-nums` is updated whenever a variable is introduced.

Unfortunately, there is no way to know during the parse phase what variables are bound by a `with` expression: These variables come from the type of the module expression, which is not known until the module expression is

[1]This restriction is articulated in the kinding rule for description variables (Appendix A, page 55) and in the typing rule for ordinary variables (Appendix A, page 58).

type checked. To fix this problem, the parser builds a `with` node where only the module expression has been parsed. (The lexical environment for the module expression is apparent.) The unparsed text of the `with` expression body and the current `alpha-env` are placed in the node. When the type of the `with` node is computed, the parsing continues with appropriate identifiers added to the `alpha-env`. Identifiers introduced in the body of the `with` expression are still numbered properly since the `cur-nums` component of the `alpha-env` stored in the `with` node is updated by side effect.

3.2 Representing Descriptions

The implementation of [Jouvelot and Gifford 88] did not distinguish the abstract and concrete syntax of descriptions: Descriptions were represented in the implementation with the text used to write them in programs. This has disadvantages:

- Code in the type checker relies unnecessarily on the concrete syntax of the language. Changes to the syntax then have wide ranging effects.
- Error messages and other code which prints descriptions does not unparse descriptions. This is of particular concern when dependent descriptions (`dselect`) appear in the language since these expressions would naturally contain parse trees representing value expressions. Keeping the parse tree in a `dselect` description means that code which does substitutions (see the next section) can use the same abstractions as all other code which manipulates programs after the parse phase.
- There is no clean way to associate particular information with descriptions, *e.g.*, their kinds, without parse trees.

Clearly there are *ad hoc* solutions to the last two problems, but all the problems are cleanly solved by creating an abstract syntax for descriptions and implementing a parse tree similar to the one for expressions beneath the abstraction. To do this, the present implementation contains a parser and an unparser for descriptions as well as all the appropriate accessor functions for the abstract nodes. All code in the implementation handling descriptions had to be changed to use the abstractions.

3.3 Supporting Substitutions

The `dstore` in the original implementation was a substitution environment mapping description variables to their definitions. The definitions in the `dstore` were *fully reduced*: free variables were substituted away, β and η reductions were completed, and recursive types were represented with circular structures.

The new design does not have the transparent recursive descriptions of *FX-87*: recursion in the description domain is only allowed through abstractions in modules. Thus, descriptions in this language have normal forms.

The new design requires that the actual text of the definition of every description and ordinary variable be kept. An occurrence of any variable in a dependent description can have its definition substituted for it when leaving the scope of an application, `let`, *etc.* However, the fully reduced definitions are also useful for cases where the substitution happens before something is type checked, *e.g.* in the `plet` rule.

To accommodate these needs, the current implementation extends the `dstore` so that every variable maps to two things:

- Its definition in normal form. (Because comparisons of expressions in descriptions rely on the expressions' textual definition. Therefore, there is no need to store a anything here for ordinary variables.)
- The full text of its definition.

The *FX-87* kind rules did not need to call the type checker and, in fact, did not need to do any substitutions of description variables. It therefore did not need the `dstore`. Since the new kind checker does invoke the type checker, the `dstore` has to be passed around in the kind checker. The only routine which makes use of the `dstore` is `kind-of-dselect`, which needs it to invoke the type checker on a module expression.

3.4 Input From Separate Files

There are several ways to support the assumption made on page 32 that the file system is immutable. The current system stamps every `input` expression with a tag derived from the universal time of the last file update. This tag is compared whenever two dependent descriptions containing `input` expressions are compared.

This is not enough by itself because a file may hide the fact that it imports from another file. It merely places the `input` inside an abstraction.

There are several ways to handle this problem:

- Allow only compiled files in `input` expressions. In order for a file to make use of a new version of a file it imports, it must be recompiled. This will alter the time of last update for the object file.
- Force the user to specify more information, *e.g.* a stamp in the form of the universal time of the last file write, or a version number (assuming version numbers are uncorruptible).
- Infer the above sort of identifying information. The stamp assigned to the `input` of a file may be the max of its universal time stamp and the stamps of the files it `input`s.[2]
- Compute a checksum for the `input` file and the files it `input`s. This method is probabalistic, but it permits a user to recompile a file without fear that the entire system will need to be recompiled.

The current implementation takes the first approach because it is the simplest. Speeding up the speed of compilation or adopting the last approach would make interactive program development easier. Incremental compilation schemes for *FX* are the subject of current research.

[2] This approach is inspired by the implementation of the sharing mechanism in ML [MacQueen 88].

3.5 Run Time Support

A module is represented at run-time as a record with fields corresponding to the module's value bindings. The precise representation interacts with the choice of subtyping rule for `modof` types. If there were no subtyping rule or if subtyping were only by prefix, then modules could easily be implemented as vectors and field selections could be translated into vector references with constant indices. If subtyping were only by field reordering (not by supersetting), then fields could be kept in a canonical order, alphabetical order say, and the same vector implementation would work.

Using a vector implementation in the presence of the more general `modof` subtyping rules of Chapter 2 implies generating additional code when matching two `modof` types requires subsetting. Assuming that fields are stored in some canonical order, the code can either generate a structure for subsequent selections to indirect through, or the code can copy the vector to a smaller vector. ML copies the module at run-time in a process called *thinning* [MacQueen 88].

For simplicity, this implementation represents modules as association lists even though access time is proportional to the number of fields in the module. The value bindings of a `mod` expression are already in the right form, and accesses to module fields just become `assq` expressions.

`Up-` and `down-` subroutines simply compile to identity functions. It is a relatively simple matter, when programs are alpha-renamed, for an implementation to detect when an `up-` or `down-` call refers to a coercion subroutine supplied as part of the implementation of a `mod` expression. In such a case, the call can be removed. The only time the identity function really needs to exist is when the programmer passes it out of the module or into another subroutine.

3.6 The `fx` Module

As an experiment to test the utility of SDTs, most of the built-in data types were removed from the *FX-87* implementation. Instead, when the new *FX* system is loaded, it automatically inserts a definition of `fx` into the environment whose value is the result of loading the `fx` module from a file.

The `fx` module is itself written in *FX*. It loads a set of modules, one for each library type, and repackages them into one module. Here is part of its definition:

```
(let ((refof-mod  (input "PSRG:FX;IMPL88;REFOF.FXBIN"))
      (int-mod    (input "PSRG:FX;IMPL88;INT.FXBIN"))
      (char-mod   (input "PSRG:FX;IMPL88;CHAR.FXBIN"))
      (string-mod (input "PSRG:FX;IMPL88;STRING.FXBIN"))...)
  (let* ((char-mod   (char-mod int-mod))
         (string-mod (string-mod int-mod char-mod refof-mod))
         ...)
    (mod
     ((unit type refof-mod.unit)
      (refof (dfunc (type region) type)
             refof-mod.refof)
      (int type int-mod.int)
      ...)
     ((new
       (poly ((r region))
         (poly ((t type))
           (subr (alloc r) (t)
             (refof t r))))
       (plambda ((r region))
         (plambda ((t type))
           (lambda ((v t))
             (up-refof
               ((proj (proj refof-mod.new r) t) v))))))
      ...
      (set
       (poly ((r region))
         (poly ((t type))
           (subr (alloc r) ((refof t r) t)
             unit)))
       (plambda ((r region))
         (plambda ((t type))
           (lambda ((x (refof t r)) (v t))
         (up-unit ((proj (proj refof-mod.set r) t)
                   (down-refof x) v))))))
    (= (subr pure (int int)
         bool)
       (lambda ((x int) (y int))
         (int-mod.= (down-int x) (down-int y))))
    ...))))
```

Notice that some modules are abstracted over others. In particular, the string module is abstracted over both the integer and character modules.

There are special provisions for constants of library types. The implementation knows that there will be an `fx` module, and it assumes that the code generation of constants is compatible with the implementation. Therefore, it assigns the type `fx.`t to a constant of type t.

3.7 Evaluation of the Implementation

It may be possible to make the implementation much more efficient by exploiting sharing in type checker structures the way ML does [MacQueen 88]. Substitutions currently take a fair amount of time. There is also a lot of complexity in the implementation to support the circular structures representing *FX-87* recursive types. But all this mechanism is no longer necessary because there are no transparent recursive descriptions.

SDTs seem to have enough power to express program modularity and simplify program linking. However, writing the `fx` module and other sample programs revealed several avenues for future improvement. Section 4.1 discusses this further.

Chapter 4

Summary

4.1 Directions for Future Research

4.1.1 Bounded Quantification

Consider the identity function on some type t. Its type is $t \rightarrow t$. If a programmer calls such a function with a value of type $t' \sqsubseteq t$, the result is still of type t.[1] The system does not provide a way to say that the return type is the same as the input type, whatever that is.

Bounded quantification [Cardelli 85] allows one to express the idea that a value's type is preserved. With universal bounded quantification, a programmer could write the identity function of type: $\forall t' \leq t.t' \rightarrow t'$. Then, when the programmer projects this function onto a particular t', the return type will be t'. Quest and CLU (with `where` clauses) provide bounded quantification.

[Lucassen 87] discusses bounded quantification in conjunction with *FX*. It seems a very useful feature, and adding it to *FX* with SDTs presents no problem. Quest provides bounded quantification in conjunction with existential quantification. This would also be no problem in the context of SDTs, but it is unclear that it provides any benefit over the ability to do upward type coercions (as with `the`).

[1]In fact, in a system with implicit subtyping at subroutine calls, `the` is just a syntactic sugar for a call to the polymorphic identity function.

4.1.2 Transparent Descriptions

The next version of *FX* will have a module system allowing the export of abstraction, value, and *transparent* description bindings. These bindings appear in the module and in the type. For example, the following `modof` type exports an abstract type `a`, a transparent type `t` whose definition is given, a value of the abstract type, and a subroutine which accepts an argument of the transparent type.

```
(modof ((a type))
       ((t (pairof a a @red)))
       ((x a)
        (f (subr pure ((y t)) a))))
```

Since module interface types are often quite long, shorthand names are very useful. A good convention might be that the author of a module create a separate file which contains the module's interface packaged as a transparent binding in another module. Then, programmers who want to specify the interface type in the argument to a subroutine can load the interface file and select the transparent bindings out.

```
(let ((interface (input "fx-interface.fx")))
  (lambda ((fx-impl interface.fx))
    ...run FX programs...))
```

The question arises whether the transparent descriptions should be recursive. If so, should they be similar to the recursive descriptions of *FX-87*? Should the abstractions be able to refer to the transparent names? There are a host of such questions currently under consideration.

4.1.3 Local Bindings and Syntax

The current system make no provision for local bindings. One can get the effect of local bindings by defining a module with all the fields required and then embed the module in a `the` form and coerce it to a type that excludes the bindings which were intended to be local:

```
(the (modof ((a type))
            ((x a)))
  (mod ((a type int)
        (b type bool))
       ((x a (up-a 3))
        (y b (up-b #t)))))
```

This scheme is verbose and requires duplicated declaration information.

One alternative is to provide bindings which are labeled as local explicitly. Then, those bindings do not appear in the type of the module. One must be careful that a local name does not appear free in a transparent definition or in a field type.

```
(mod ((a type int)
      (local b type bool))
      ((x a (up-a 3))
       (local y b (up-b #t))))
```

Another approach is to provide an explicit export list where the programmer provides a list of exported names. This approach assumes bindings are local unless specifically exported whereas the previous approach assumes the opposite. Two alternative syntaxes are:

```
(mod ((a type int)
      (b type bool))
      ((x a (up-a 3))
       (y b (up-b #t)))
  (exports a x))
```

```
(exports a x
  (mod ((a type int)
        (b type bool))
       ((x a (up-a 3))
        (y b (up-b #t)))))
```

The module syntax, as currently implemented, is currently too verbose. There are several approaches to a cleaner syntax. The interface types could all be moved to a separate part of the module, so that the interface information is all together and the implementation information is all together. Perhaps the argument types to subroutines may be omitted since they appear in the interface. The same should be true of kinds of polymorphic values.

These syntactic problems may lessen with the introduction of type reconstruction into *FX* [O'Toole and Gifford 89]. Type reconstruction allows the compiler to deduce declarations omitted by the programmer.

4.1.4 Opening Modules

One problem, illustrated clearly by the `fx` module itself, is the way fields inherited from a module must be repackaged into a new module. An `open` form which makes all the names of a module available in some scope would be most useful. There are quite a few design issues with such a facility: Do the names shadow other names in the current environment? Is there an elegant way to do systematic renaming?

It would also be useful to have a more general facility for converting the types of values to use abstract names. Currently, a subroutine which uses the representation of some abstraction cannot be converted directly to a subroutine using the abstraction. Instead, as in the `fx` module, the arguments (description and value) must be collected and converted to the representation type, the subroutine must be called on the converted values, and the result must be converted to the abstract type. This is unnecessary work. In addition, such convoluted code is error prone and adds extra subroutine calls which may be difficult for the compiler to open code.

There may be a way to solve the above problem together with the problem of merging modules (a common operation). The form `(extend a b)` might the fields of the modules `a` and `b`, changing the types of the fields to reflect the re-exported abstractions. Problems of name clashes and renaming remain.

4.1.5 Sharing

ML has a mechanism for describing and enforcing sharing constraints. The following example comes from [MacQueen 88].

```
signature SYMBOL = sig type symbol ... end

signature LEX =
 sig
```

```
  structure Symbol : SYMBOL
  val next : unit -> Symbol.symbol
  ...
 end

signature SYMBOLTABLE =
 sig
  structure Symbol : SYMBOL
  type var
  val  bind : Symbol.symbol * var -> unit
  ...
 end

signature PARSE_ARGS =
 sig
  structure Lex : LEX
  structure SymTab : SYMBOLTABLE
  sharing   Lex.Symbol = SymTab.Symbol
 end

functor Parse (A : PARSE_ARGS) =
 struct ... A.SymTab.bind(A.Lex.next(), v) ... end
```

In this example, the ML system guarantees that `Lex.Symbol` and `SymTab.Symbol` are the same structure. This will allow the module created by a call to `Parse` to pass the symbol values returned by `Lex.next` to `SymTab.bind`.

The system presented in this thesis has no concise way to express sharing constraints. The programmer can abstract the `LEX` and `SYMBOLTABLE` modules over their submodules (`Symbol` in this case). Then the `PARSE_ARGS` constructor can take a `Symbol` module as an argument and explicitly pass it to the `Lex` `SymTab` modules. This is awkward and entails extra subroutine calls which may be difficult for the compiler to open code.

The problem with the ML sharing mechanism is that it is based on a particular *ad hoc* algorithm for assigning tags to structures. This is, in fact, the inspiration for the method of implementing `input` described in Chapter

3. However, the programmer does not need to know about the tags in the implementation of `input`: the implementation supports simple abstraction of an immutable file system articulated in the design. An approach to sharing based on some higher level abstraction would be valuable.

4.1.6 Persistence

One important restriction of the system presented here is that the type of a value in a file must be known statically. This does not support persistent typed values. Quest has some support for this in the form of `automatic` types, and CLU provides the `any` type. An automatic value has its actual type attached to its run time representation, and its exact type is not known statically. The programmer may convert any value to an automatic value but can only access an automatic value through a case dispatch on its type (like a case discrimination for a tagged union type with the full generality of the type system in the tag). The system tests to see if the type tag is equivalent to a type in the case expression at run time. This makes highly optimized type equivalence checking and type representation important. Adding such a facility to *FX* is not difficult, but the implementation costs are not yet clear.

One fairly simple way to get persistence is to put such tagged values in files. Then, the language could supply the form `(load T E)` where `T` is a type and `E` is an expression which computes a string. At run time, the file named in the string computed by `E` can be read, and its type tag can be compared dynamically with `T`. Computation proceeds only if the types are equivalent.

4.2 Summary

Static dependent types support a module system which is more flexible than other statically type checked systems. This flexibility has proven both practical and useful in an implementation built on the *FX-87* Interpreter; in particular, the module system provides a uniform way to build small program structures and to link them together into a complete system. Static effect constraints guarantee that the type system is sound in the presence of side effects. Though work remains to make the notation for modules more compact, the essential system of static dependent types is a promising approach to the construction of large systems.

Appendix A

An FX-87 Kernel Language

The following kernel language is a subset of the *FX-87* kernel and forms the basis for the extensions described in the thesis. This language differs from *FX-87* in the following ways:

- There are no mutable variables.
- There is no implicit subtyping. Programmers must write `the` forms to do explicit upward type coercions.
- There is a new `does` form for doing upward effect coercion; `the` no longer performs effect coercions.
- Recursion in both the description and value domains is accomplished using the module proposal in the thesis. This implies the demise of transparent recursive descriptions (*i.e.*, that recursive descriptions no longer admit structural equality). They would be easy to add again and their design is orthogonal to the issues examined in the thesis.

A.1 Syntax

A.1.1 Kinds

Kexp = – Kinds (κ)

`region` | `effect` | `type`
(`dfunc` (*Kexp*...) *Kexp*)

A.1.2 Descriptions

Dexp = – Descriptions (δ)

Rexp | *Eexp* | *Texp* | *HDesc*

Rexp = – Regions (ρ)

`@`*id*
(`runion` *Rexp* *Rexp* ...)
GDesc

Eexp = – Effects (ϵ)

`pure`
(`alloc` *Rexp*)
(`read` *Rexp*)
(`write` *Rexp*)
(`maxeff` *Eexp*...)
GDesc

Texp = – Types (τ)

`bool`
(`subr` *Eexp* (*Texp*...) *Texp*)
(`poly` ((*var* *Kexp*)...) *Texp*)
GDesc

HDesc = – Higher order descriptions

    (`dlambda` ((*id Kexp*)...) *Dexp*)
    GDesc

GDesc = – Generic descriptions

    id
    (*HDesc Dexp*...)
    (`dlet` ((*id Dexp*)...) *Dexp*)

A.1.3 Value Expressions

Exp = – Expressions (e)

    id
    (`lambda` ((*id Texp*)...) *Exp*)
    (*Exp Exp*...)
    (`let` ((*id Exp*)...) *Exp*)
    (`if` *Exp Exp Exp*)
    (`begin` *Exp Exp*...)
    (`plambda` ((*id Kexp*)...) *Exp*)
    (`proj` *Exp Dexp*...)
    (`plet` ((*id Dexp*)...) *Exp*)
    (`the` *Texp Exp*)
    (`does` *Eexp Exp*)
    Literal

Literal = – Literals

    `#t` | `#f`

A.2 Static Semantics

The *FX-87* kind inference rules use a kind assignment, and the type inference rules use a type assignment. The kind and type environments introduce two distinct namespaces which must be kept consistent in order to preserve the single namespace semantics of *FX*.[1] Because of this problem, and because kind inference in the dependent description system described in the thesis requires both type assignment and kind assignment information, the rules given here are written with a single environment (TK) for both sorts of bindings.

A.2.1 Kind Inference Rules

The following rules form an inductive definition of the *has kind* (::) relation.

There is an infinite set of region constants which begin with an `@`. The programmer chooses these names.

There is one distinguished region `@=` which is the *immutable* region. References stored in the mutable region may not be stored into.

An expression with no side effects is said to be `pure`.

The type of the true and false values is `bool`.

$$\begin{array}{rcl} \mathrm{TK} \vdash @\mathit{id} & :: & \texttt{region} \\ \mathrm{TK} \vdash \texttt{pure} & :: & \texttt{effect} \\ \mathrm{TK} \vdash \texttt{bool} & :: & \texttt{type} \end{array}$$

The type and kind environment (TK) maps bound description identifiers to their kinds. The restriction that a given identifier may not be rebound does not constrain programs if the implementation does global alpha-renaming. (See Section 3.)

$$\frac{\mathit{id} \notin \mathrm{Domain}(\mathrm{TK})}{\mathrm{TK}[\mathit{id} :: \kappa] \vdash \mathit{id} :: \kappa}$$

[1]Pierre Jouvelot pointed out this shortcomming of the semantics in [Gifford, *et al.* 87].

The `runion` region constructor forms the set union of the given regions.

$$\frac{\mathrm{TK} \vdash \rho_i :: \texttt{region} \quad (1 \leq i \leq n)}{\mathrm{TK} \vdash (\texttt{runion}\ \rho_1\ \rho_2 \ldots) :: \texttt{region}}$$

The `alloc` form indicates an allocation *and* initialization of a location in some region of the store.

$$\frac{\mathrm{TK} \vdash \rho :: \texttt{region}}{\mathrm{TK} \vdash (\texttt{alloc}\ \rho) :: \texttt{effect}}$$

The `read` form indicates a dereferencing of a location in some region of the store.

$$\frac{\mathrm{TK} \vdash \rho :: \texttt{region}}{\mathrm{TK} \vdash (\texttt{read}\ \rho) :: \texttt{effect}}$$

The `write` form indicates a mutation of a location in some region of the store.

$$\frac{\begin{array}{r}\mathrm{TK} \vdash \rho :: \texttt{region}\\ \mathrm{Mutable}(\rho)\end{array}}{\mathrm{TK} \vdash (\texttt{write}\ \rho) :: \texttt{effect}}$$

The Mutable predicate is true of every region except the immutable region `@=` and any region containing it. The Immutable predicate is the negation of the Mutable predicate.

$$\mathrm{Immutable}(\rho) \text{ iff } \texttt{@=} \sqsubseteq \rho$$

Forbidding `write` effects on the immutable region prevents any reference stored there from being stored into.

The `maxeff` form represents the combination of a set of effects.

$$\frac{\mathrm{TK} \vdash \epsilon_i :: \texttt{effect} \quad (1 \leq i \leq n)}{\mathrm{TK} \vdash (\texttt{maxeff}\ \epsilon_1 \ldots) :: \texttt{effect}}$$

The type of a subroutine includes the effect incurred by an application of a subroutine value with that type (the *latent effect*), as well as the types of the arguments and of the return value. (This rule will be replaced in section 2.2.1.)

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \epsilon & :: & \texttt{effect} \\ \mathrm{TK} \vdash \tau_i & :: & \texttt{type} \quad (1 \leq i \leq n) \\ \mathrm{TK} \vdash \tau & :: & \texttt{type} \end{array}}{\mathrm{TK} \vdash (\texttt{subr}\ \epsilon\ (\tau_1 \ldots)\ \tau) :: \texttt{type}}$$

A `poly` form is the type of a value which is polymorphic over some descriptions whose kinds are given in the type.

$$\frac{\mathrm{TK}[_{i=1}^{n} id_i :: \kappa_i] \vdash \tau :: \texttt{type}}{\mathrm{TK} \vdash (\texttt{poly}\ ((id_1\ \kappa_1) \ldots)\ \tau) :: \texttt{type}}$$

Users may define their own description functions (*e.g.*, type and effect constructors) with the `dlambda` form.

$$\frac{\mathrm{TK}[_{i=1}^{n} id_i :: \kappa_i] \vdash \delta :: \kappa}{\mathrm{TK} \vdash (\texttt{dlambda}\ ((id_1\ \kappa_1) \ldots)\ \delta) :: (\texttt{dfunc}\ (\kappa_1 \ldots)\ \kappa)}$$

Combinations in descriptions represent applications of description functions.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \delta & :: & (\texttt{dfunc}(\kappa_1 \ldots)\kappa) \\ \mathrm{TK} \vdash \delta_i & :: & \kappa_i \quad (1 \leq i \leq n) \end{array}}{\mathrm{TK} \vdash (\delta\ \delta_1 \ldots) :: \kappa}$$

The `dlet` form is a way to introduce local description synonyms. The result is the same as having written the definitions in place of the local names in the body. (See the equivalence rule for `dlet` below.)

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \delta_i & :: & \kappa_i \\ \mathrm{TK}[^n_{i=1} id_i :: \kappa_i] \vdash \delta & :: & \kappa \end{array}}{\mathrm{TK} \vdash (\texttt{dlet}\ ((id_1\ \delta_1) \ldots)\ \delta) :: \kappa}$$

A.2.2 Type and Effect Inference Rules

The *has type* (**:**) and the *has effect* (**!**) relations hold up to description equivalence (defined below in section A.2.3):[2]

$$\frac{\begin{array}{c} \mathrm{TK} \vdash e : \tau \ ! \ \epsilon \\ \tau \equiv \tau' \quad \wedge \quad \epsilon \equiv \epsilon' \end{array}}{\mathrm{TK} \vdash e : \tau' \ ! \ \epsilon'}$$

Moreover, the **!** relation is not unique because of *effect masking*:[3] If a region constant or region variable appears free in the effect of an expression but does not appear free in the type of any free variable of the expression, then any `read` or `write` effects on that region may be masked; furthermore, if the region constant or variable does not appear free in the type of the expression, then any `alloc` effect on the region may be masked.

Suppose the set of effect constants is extended by the distinguished region o denoting a region with no locations. Any effects on o are interconvertible with `pure`, and (`runion` o ρ) is equivalent to ρ. The following rules provide a formal definition of effect masking:

[2]Pierre Jouvelot suggested the following rule as a compact formalization of this property.

[3]The following description of effect masking, including the inference rules, is adapted (by minor rewording and simple notational variations) from [Gifford, *et al.* 87, pp. 132–133].

$$\frac{\begin{array}{rrrcl} & & \mathrm{TK} \vdash e & : & \tau \ \texttt{!} \ \epsilon \\ & & \mathrm{TK} \vdash \delta & :: & \texttt{region} \\ (id \in \mathrm{FV}(e)\ ) & \wedge & (\mathrm{TK} \vdash id : \tau') & \Rightarrow & \delta \notin \mathrm{FV}(\tau') \cup \mathrm{FRC}(\tau') \end{array}}{\mathrm{TK} \vdash e \ \texttt{!} \ (\texttt{maxeff} \ [o/\delta]\epsilon \ (\texttt{alloc} \ \delta))}$$

$$\frac{\begin{array}{rrrcl} & & \mathrm{TK} \vdash e & : & \tau \ \texttt{!} \ \epsilon \\ & & \mathrm{TK} \vdash \delta & :: & \texttt{region} \\ (id \in \mathrm{FV}(e)\ ) & \wedge & (\mathrm{TK} \vdash id : \tau') & \Rightarrow & \delta \notin \mathrm{FV}(\tau') \cup \mathrm{FRC}(\tau') \\ & & \delta & \notin & \mathrm{FV}(\tau) \end{array}}{\mathrm{TK} \vdash e \ \texttt{!} \ [o/\delta]\epsilon}$$

FV is the function which returns the set of free variables of an expression or description. FRC is the function which returns the set of free region constants of an expression.

This implies that an expression may have more than one effect. By convention, any assertion of the form $e \ \texttt{!} \ \epsilon$ in a *premise* of a typing rule will mean that ϵ is the *least* effect of e under TK. The partial order under which the effect is least is the one defined by the description inclusion relation $\sqsubseteq$ given below in Section A.2.3.

The kernel contains boolean constants for *true* and *false*:

$$\begin{array}{lcl} \mathrm{TK} \vdash \texttt{\#t} & : & \texttt{bool} \ \texttt{!} \ \texttt{pure} \\ \mathrm{TK} \vdash \texttt{\#f} & : & \texttt{bool} \ \texttt{!} \ \texttt{pure} \end{array}$$

The type and kind environment maps bound value identifiers to their types. Looking up a variable is always `pure` since there are no mutable variables.

$$\frac{id \notin \mathrm{Domain}(\mathrm{TK})}{\mathrm{TK}[id : \tau] \vdash id : \tau \ \texttt{!} \ \texttt{pure}}$$

The `lambda` form is the constructor of subroutine values. Notice that the latent effect is encoded in the subroutine type. (This rule will be replaced in section 2.2.2.)

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \tau_i & :: & \texttt{type} \quad (1 \leq i \leq n) \\ \mathrm{TK}[^n_{i=1} id_i \texttt{ : } \tau_i] \vdash e & \texttt{:} & \tau \texttt{ ! } \epsilon \end{array}}{\mathrm{TK} \vdash (\texttt{lambda } ((id_1\ \tau_1) \ldots)\ e) \texttt{ : } (\texttt{subr } \epsilon\ (\tau_1 \ldots)\ \tau) \texttt{ ! pure}}$$

Combinations represent subroutine application. The type of an application is the return type of the subroutine. The effect of an application is the combined effect of evaluating the subroutine value and all the arguments *and* the latent effect of the subroutine.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e & \texttt{:} & (\texttt{subr } \epsilon_l\ (\tau_1 \ldots)\ \tau) \texttt{ ! } \epsilon \\ \mathrm{TK} \vdash e_i & \texttt{:} & \tau_i \texttt{ ! } \epsilon_i \quad (1 \leq i \leq n) \end{array}}{\mathrm{TK} \vdash (e\ e_1 \ldots) \texttt{ : } \tau \texttt{ ! } (\texttt{maxeff } \epsilon_l\ \epsilon_1 \ldots)}$$

A `let` expression could be rewritten (with suitable typing information) as an application of a `lambda` expression. But since the argument types are easily deducible from the `let` bindings, this form is included in the kernel.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e_i & \texttt{:} & \tau_i \texttt{ ! } \epsilon_i \quad (1 \leq i \leq n) \\ \mathrm{TK}[^n_{i=1} id_i \texttt{ : } \tau_i] \vdash e & \texttt{:} & \tau \texttt{ ! } \epsilon \end{array}}{\mathrm{TK} \vdash (\texttt{let } ((id_1\ e_1) \ldots)\ e) \texttt{ : } \tau \texttt{ ! } (\texttt{maxeff } \epsilon\ \epsilon_1 \ldots)}$$

The two arms of a conditional expression must have the same type.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e & \texttt{:} & \texttt{bool ! } \epsilon \\ \mathrm{TK} \vdash e_1 & \texttt{:} & \tau \texttt{ ! } \epsilon_1 \\ \mathrm{TK} \vdash e_2 & \texttt{:} & \tau \texttt{ ! } \epsilon_2 \end{array}}{\mathrm{TK} \vdash (\texttt{if } e\ e_1\ e_2) \texttt{ : } \tau \texttt{ ! } (\texttt{maxeff } \epsilon\ \epsilon_1\ \epsilon_2)}$$

The `begin` form is for sequencing (side-effecting) expressions. The value of a `begin` form is the value of the last expression in it.

$$\frac{\mathrm{TK} \vdash e_i \texttt{ : } \tau_i \texttt{ ! } \epsilon_i \quad (1 \le i \le n)}{\mathrm{TK} \vdash (\texttt{begin } e_1\ e_2 \ldots) \texttt{ : } \tau_n \texttt{ ! } (\texttt{maxeff } \epsilon_1 \ldots)}$$

Polymorphic values are created with the `plambda` form: an expression may be abstracted over a set of description arguments. Requiring the body of a `plambda` expression to be `pure` simplifies the type system (because `poly` types do not need a latent effect) and allows the value to evaluated once at the point of definition. Projections are, then, zero cost.

The rule given here is simpler than the corresponding rule from [Gifford, *et al.* 87, p. 129]. The condition is that free variables of the body of a `plambda` may not contain free occurrences of any of the `plambda`-bound variables. (For a thorough explanation of the restriction, see [McCracken 79, p. 20–21]) The restriction is no longer necessary because of the assumption that variable names may not be rebound in TK.

$$\frac{\mathrm{TK}[^n_{i=1} id_i \texttt{ :: } \kappa_i] \vdash e \texttt{ : } \tau \texttt{ ! pure}}{\mathrm{TK} \vdash (\texttt{plambda } ((id_1\ \kappa_1) \ldots)\ e) \texttt{ : } (\texttt{poly } ((id_1\ \kappa_1) \ldots)\ \tau) \texttt{ ! pure}}$$

To get an instantiation of a polymorphic value, one projects it onto a set of description arguments of the appropriate kinds.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash e & \texttt{:} & (\texttt{poly } ((id_1\ \kappa_1) \ldots)\ \tau) \texttt{ ! } \epsilon \\ \mathrm{TK} \vdash \delta_i & \texttt{::} & \kappa_i \quad (1 \le i \le n) \end{array}}{\mathrm{TK} \vdash (\texttt{proj } e\ \delta_1 \ldots) \texttt{ : } [^n_{i=1} \delta_i / id_i]\tau \texttt{ ! } \epsilon}$$

Just as `let` allows local value bindings, `plet` allows local description bindings. Like `dlet`, the bindings are transparent synonyms: it is just as though the definitions had been used in the body in place of the local names. Thus, `plet` is *not* like an applied `plambda`.

$$\frac{\begin{array}{rcl} \mathrm{TK} \vdash \delta_i & \texttt{::} & \kappa_i \quad (1 \le i \le n) \\ \mathrm{TK} \vdash [^n_{i=1} \delta_i / id_i]e & \texttt{:} & \tau \texttt{ ! } \epsilon \end{array}}{\mathrm{TK} \vdash (\texttt{plet } ((id_1\ \delta_1) \ldots)\ e) \texttt{ : } \tau \texttt{ ! } \epsilon}$$

A `the` expression asserts that an expression has a larger type.

$$\frac{\mathrm{TK} \vdash e \mathbin{\texttt{:}} \tau' \mathbin{\texttt{!}} \epsilon' \qquad \tau' \sqsubseteq \tau}{\mathrm{TK} \vdash (\texttt{the}\ \tau\ e) \mathbin{\texttt{:}} \tau \mathbin{\texttt{!}} \epsilon'}$$

A `does` expression asserts that an expression has a larger effect.

$$\frac{\mathrm{TK} \vdash e \mathbin{\texttt{:}} \tau' \mathbin{\texttt{!}} \epsilon' \qquad \epsilon' \sqsubseteq \epsilon}{\mathrm{TK} \vdash (\texttt{does}\ \epsilon\ e) \mathbin{\texttt{:}} \tau' \mathbin{\texttt{!}} \epsilon}$$

Note that the `the` and `does` rules are the *only* typing rules which make use of the inclusion relation (defined below). This means that all coercions must be explicit.

A.2.3 Description Inclusion

The description inclusion relation ($\sqsubseteq$) is the reflexive-transitive closure of the following rules. Two descriptions are equivalent ($\equiv$) if and only if each is included in the other.

The rule for inclusion on identifiers might suggest that description inclusion is context-dependent; however, this is not true because programs are globally alpha-renamed in the implementation described in the thesis. (See Section 3.1.)

$$id \sqsubseteq id$$

Region constants are equivalent if they have the same name. The immutable region is included in every other region.

$$@id \sqsubseteq @id$$

The `runion` operation is set union on regions. It is idempotent, commutative, and associative.

$$\begin{aligned}
(\texttt{runion}\ \rho) &\equiv \rho \\
(\texttt{runion}\ \rho_1\ \rho_2) &\equiv (\texttt{runion}\ \rho_2\ \rho_1) \\
(\texttt{runion}\ (\texttt{runion}\ \rho_1\ \rho_2)\ \rho_3) &\equiv (\texttt{runion}\ \rho_1\ (\texttt{runion}\ \rho_2\ \rho_3)) \\
(\texttt{runion}\ \rho\ \rho) &\equiv \rho
\end{aligned}$$

Region inclusion is the same as set inclusion.

$$\frac{\forall i \in [1,n] \quad \exists j \in [1,m] \text{s.t.} \rho_i \sqsubseteq \rho_j}{(\texttt{runion}\ \rho_1\ \rho_2 \ldots \rho_n) \sqsubseteq (\texttt{runion}\ \rho'_1\ \rho'_2 \ldots \rho'_m)}$$

Effects form a lattice with `pure` as bottom element; `pure` denotes the absence of all side-effects.

$$\texttt{pure} \sqsubseteq \epsilon$$

`alloc` and `read` effects on immutable regions are equivalent to `pure`.

$$\frac{\text{Immutable}(\rho)}{(\texttt{alloc}\ \rho) \equiv \texttt{pure}}$$

$$\frac{\text{Immutable}(\rho)}{(\texttt{read}\ \rho) \equiv \texttt{pure}}$$

Inclusion on primitive effects (`alloc`, `read`, `write`) is inherited from inclusion on the regions over which the effects take place.

$$\frac{\rho_1 \sqsubseteq \rho_2}{(\texttt{alloc}\ \rho_1) \sqsubseteq (\texttt{alloc}\ \rho_2)}$$

$$\frac{\rho_1 \sqsubseteq \rho_2}{(\texttt{read}\ \rho_1) \sqsubseteq (\texttt{read}\ \rho_2)}$$

$$\frac{\rho_1 \sqsubseteq \rho_2}{(\texttt{write}\ \rho_1) \sqsubseteq (\texttt{write}\ \rho_2)}$$

The `maxeff` operation on effects is also like set union. The empty `maxeff` is another name for `pure`. Like `runion`, `maxeff` is idempotent, commutative, and associative.

$$\begin{array}{rcl}
(\texttt{maxeff}) & \equiv & \texttt{pure} \\
(\texttt{maxeff}\ \epsilon) & \equiv & \epsilon \\
(\texttt{maxeff}\ \epsilon_1\ \epsilon_2) & \equiv & (\texttt{maxeff}\ \epsilon_2\ \epsilon_1) \\
(\texttt{maxeff}\ (\texttt{maxeff}\ \epsilon_1\ \epsilon_2)\ \epsilon_3) & \equiv & (\texttt{maxeff}\ \epsilon_1\ (\texttt{maxeff}\ \epsilon_2\ \epsilon_3)) \\
(\texttt{maxeff}\ \epsilon\ \epsilon) & \equiv & \epsilon
\end{array}$$

Inclusion on `maxeff`s is like inclusion on `runion`.

$$\frac{\forall i \in [1,n] \quad \exists j \in [1,m] \text{ s.t. } \epsilon_i \sqsubseteq \epsilon_j}{(\texttt{maxeff}\ \epsilon_1 \ldots \epsilon_n) \sqsubseteq (\texttt{maxeff}\ \epsilon'_1 \ldots \epsilon'_m)}$$

Subroutine types are *monotonic* in their effect and return type components, but *anti-monotonic* in their argument type components.

$$\frac{\begin{array}{rcl}\epsilon & \sqsubseteq & \epsilon' \\ \tau_i' & \sqsubseteq & \tau_i \quad (1 \leq i \leq n) \\ \tau & \sqsubseteq & \tau'\end{array}}{(\texttt{subr}\ \epsilon\ (\tau_1 \ldots \tau_n)\ \tau) \sqsubseteq (\texttt{subr}\ \epsilon'\ (\tau_1' \ldots \tau_n')\ \tau')}$$

Bound variables in `poly` expressions may be alpha-renamed:

$$\frac{id_i' \notin \mathrm{FV}(\tau) \quad (1 \leq i \leq n)}{\begin{array}{c}(\texttt{poly}\ ((id_1\ \kappa_1) \ldots (id_n\ \kappa_n))\ \tau) \\ \equiv \\ (\texttt{poly}\ ((id_1'\ \kappa_1) \ldots (id_n'\ \kappa_n))\ [^n_{i=1} id_i'/id_i]\tau)\end{array}}$$

If every projection of a polymorphic value is a subtype of every projection of another polymorphic value, then the `poly` type of the first is a subtype of the `poly` type of the second. Notice that this rule may be applied in conjunction with the above alpha-renaming rule.

$$\frac{\tau \sqsubseteq \tau'}{(\texttt{poly}\ ((id_1\ \kappa_1) \ldots (id_n\ \kappa_n))\ \tau) \sqsubseteq (\texttt{poly}\ ((id_1\ \kappa_1) \ldots (id_n\ \kappa_n))\ \tau')}$$

Bound variables in `dlambda` forms may be alpha-renamed:

$$\frac{id_i' \notin \mathrm{FV}(\delta) \quad (1 \leq i \leq n)}{\begin{array}{c}(\texttt{dlambda}\ ((id_1\ \kappa_1) \ldots (id_n\ \kappa_n))\ \delta) \\ \equiv \\ (\texttt{dlambda}\ ((id_1'\ \kappa_1) \ldots (id_n'\ \kappa_n))\ [^n_{i=1} id_i'/id_i]\delta)\end{array}}$$

Description functions admit η-conversion.

$$\frac{id_i \notin \mathrm{FV}(\delta) \quad (1 \leq i \leq n)}{(\texttt{dlambda}\ ((id_1\ \kappa_1) \ldots (id_n\ \kappa_n))\ (\delta id_1 \ldots id_n)) \equiv \delta}$$

If a description function always returns a subdescription of another description function, then it is included in that description function. Notice that this rule may be applied in conjunction with the above alpha-renaming rule.

$$\frac{\delta \sqsubseteq \delta'}{(\texttt{dlambda}\ ((id_1\ \kappa_1)\ldots(id_n\ \kappa_n))\ \delta) \sqsubseteq (\texttt{dlambda}\ ((id_1\ \kappa_1)\ldots(id_n\ \kappa_n))\ \delta')}$$

An applied description function is equivalent to its body with the arguments substituted for the formals.

$$((\texttt{dlambda}\ ((id_1\ \kappa_1)\ldots(id_n\ \kappa_n))\ \delta)\delta_1\ldots\delta_n) \equiv [_{i=1}^{n}\delta_i/id_i]\delta$$

An otherwise unreducible description application is equivalent to another iff all components are equivalent. There is no proper inclusion on description applications because there is no way to say whether a description function is monotonic, anti-monotonic, or neither in an argument.

$$\frac{\delta_i \equiv \delta_i' \quad (1 \leq i \leq n)}{(desc_1 \ldots \delta_n) \equiv (desc_1' \ldots \delta_n')}$$

`dlet` forms introduce local description abbreviations. The abbreviations are transparent in the sense that they are equivalent to their definitions.

$$(\texttt{dlet}\ ((id_1\ \delta_1)\ldots)\ \delta) \equiv [_{i=1}^{n}\delta_i/id_i]\delta$$

Bibliography

[Atkinson 78] Russell R. Atkinson, Barbara H. Liskov, Robert W. Scheifler. Aspects of Implementing CLU. ACM 1978 Annual Conference Proceedings, December 1978.

[Berendregt 84] H. P. Berendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, 1984.

[Burstall and 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 85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *Computing Surveys,* Vol. 17, No. 4, pp. 471–522, December 1985.

[Cardelli 89] Luca Cardelli. Typeful Programming. IFIP Advanced Seminar on Formal Description of Programming Concepts. 1989.

[DoD 83] DoD (U.S. Department of Defense). *Ada Reference Manual.* ANSI/MIS-STD 1815 (Jan.). U.S. Government Printing Office. 1983.

[Gifford and Lucassen 86] David K. Gifford and John M. Lucassen. Integrating Functional and Imperative Programming. Proceedings of the 1986 ACM Conference on Lisp and Functional Programming. Pages 28–38.

[Gifford, *et al.* 87] David K. Gifford, Pierre Jouvelot, John M. Lucassen, Mark A. Sheldon. *FX-87* Reference Manual. MIT/LCS/TR-407. September 1987.

[Girard 71] J.-Y. Girard. Une extension de l'interprétation de Gödel à l'analyse et son application à l'élimination des coupures dans l'analyse et la théorie des types. In *Proceedings of the Second Scandinavian Logic Symposium* (edited by J. E. Fenstad). North-Holland, 1971.

[Girard 72] J.-Y. Girard. *Interpretation Fonctionelle et Elimation des Coupures dans l'Arithmetique d'Ordre Superieur.* These de Doctorat d'etat, University of Paris.

[Hammel and Gifford 88] R. Todd Hammel and David K. Gifford. *FX-87* Performance Measurements: Dataflow Implementation. MIT/LCS/TR-421. November 1988.

[Hook and Howe 86] James G. Hook and Douglas J. Howe. Impredicative Strong Existential Equivalent to Type:Type. Department of Computer Science, Cornell University, Ithaca, New York, TR 86-760, June 1986.

[Jouvelot and Gifford 88] Pierre Jouvelot and David K. Gifford. The *FX-87* Interpreter. In *Proceedings of the 2nd Int'l. Conf. on Comp. Lang.* Miami FL. October, 1988.

[Jouvelot and Gifford 89a] Pierre Jouvelot and David K. Gifford. Communication effects for message-based concurrency. MIT/LCS/TM-???, 1989.

[Jouvelot and Gifford 89] Pierre Jouvelot and David K. Gifford. Reasoning about Continuation with Control Effects. In Proceedings of the SIGPLAN PLDI'89 Conference. ACM, Portland, June 1989.

[Liskov, *et al.* 81] Barbara Liskov, Russell Atkinson, Toby Bloom, Eliot Moss, J. Craig Schaffert, Robert Scheifler, Alan Snyder. Springer-Verlag, 1981.

[Lucassen 87] John M. Lucassen. *Types and Effects: Towards the Integration of Functional and Imperative Programming.* Ph.D. Thesis. MIT/LCS/TR-408, August 1987.

[Lucassen and Gifford 88] John M. Lucassen and David K. Gifford. Polymorphic Effect Systems. in POPL 88.

[MacQueen 84] David MacQueen. Modules for Standard ML. Proceedings ACM Symposium on Lisp and Functional Programming, 1984.

[MacQueen 88] David MacQueen. An Implementation of Standard ML Modules. Proceedings ACM Symposium on Lisp and Functional Programming, 1988.

[Martin-Löf 73] An intuitionistic Theory of Types: Predicative Part. In *Logic Colloq. '73* (eds. H. E. Rose and J. C. Shepherdson) North-Holland, 73–118.

[McCracken 79] Nancy Jean McCracken. *An Investigation of a Programming Language with a Polymorphic Type Structure.* Dissertation. Syracuse University. 1979.

[Meyer and Reinhold 86] Albert R. Meyer and Mark B. Reinhold. 'Type' Is Not a Type: Preliminary Report. ACM POPL 1986.

[Mitchell and Plotkin 88] John C. Mitchell and Gordon D. Plotkin. Abstract Types Have Existential Type in *ACM Trans. on Programming Languages and Systems*, Vol. 10, No. 3, July 1988. (Published earlier in POPL 86.)

[O'Toole 89] James William O'Toole Jr., *Polymorphic Type Reconstruction*, S.M. Thesis, Massachusetts Institute of Technology, May 1989.

[O'Toole and Gifford 89] James William O'Toole Jr. and David K. Gifford, *Type Reconstruction with First-Class Polymorphic Values*, *SIGPLAN Programming Language Design and Implementation*, June 1989.

[Rees 86] Jonathan Rees, *et al.* *Revised^3 Report on the Algorithmic Language Scheme.* AI Memo 848a. MIT, Artificial Intelligence Laboratory, September 1986.