Optimizing ML Using a Hierarchy of Monadic Types

Andrew Tolmach*

Pacific Software Research Center
Portland State University & Oregon Graduate Institute
Dept. of Computer Science, P.S.U., P.O. Box 751, Portland, OR 97207, USA
apt@cs.pdx.edu

Abstract. We describe a type system and typed semantics that use a hierarchy of monads to describe and delimit a variety of effects, including non-termination, exceptions, and state, in a call-by-value functional language. The type system and semantics can be used to organize and justify a variety of optimizing transformations in the presence of effects. In addition, we describe a simple monad inferencing algorithm that computes the minimum effect for each subexpression of a program, and provides more accurate effects information than local syntactic methods.

1 Introduction

Optimizers are often implemented as engines that repeatedly apply improving transformations to programs. Among the most important transformations are *propagation* of values from their defining site to their use site, and *hoisting* of invariant computations out of loops. If we use a pure (side-effect-free) language based on the lambda calculus as our compiler intermediate language, these transformations can be neatly described by the simple equations for beta-reduction

(Beta) $$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 = e_2[e_1/x]$$

and for the exchange and hoisting of bindings

(Exchange)
$$\begin{array}{l}\texttt{let } x_1 \texttt{ = } e_1 \texttt{ in (let } x_2 \texttt{ = } e_2 \texttt{ in } e_3\texttt{)} = \\ \texttt{let } x_2 \texttt{ = } e_2 \texttt{ in (let } x_1 \texttt{ = } e_1 \texttt{ in } e_3\texttt{)} \\ \qquad (x_1 \notin FV(e_2); x_2 \notin FV(e_1))\end{array}$$

(RecHoist)
$$\begin{array}{l}\texttt{letrec } f\ x \texttt{ = (let } y \texttt{ = } e_1 \texttt{ in } e_2\texttt{) in } e_3 = \\ \texttt{let } y \texttt{ = } e_1 \texttt{ in (letrec } f\ x \texttt{ = } e_2 \texttt{ in } e_3\texttt{)} \\ \qquad (x, f \notin FV(e_1); y \notin FV(e_3))\end{array}$$

where $FV(e)$ is the set of free variables of e. The side conditions nicely express the data dependence conditions under which the equations are valid. Either

* Supported, in part, by the US Air Force Materiel Command under contract F19628-93-C-0069 and by the National Science Foundation under grant CCR-9503383.

orientation of the equation generates a valid transformation.[1] Effective compilers for pure, lazy functional languages (e.g., [11]) have been conceived and built on the basis of such transformations, with considerable advantages for modularity and correctness.

It would be nice to apply similar methods to the optimization of languages like ML, which have side effects such as I/O, mutable state, and exceptions. Unfortunately, these "rearranging" transformations are not generally valid for such languages. For example, if we apply (Beta) (oriented left-to-right) in a situation where evaluating e_1 performs output and x is mentioned twice in e_2, evaluating the resulting expression might produce the output twice. In fact, once an eager evaluation order is fixed, even non-termination becomes a "side effect." For example, (RecHoist) is not valid unless e_1 is known to be terminating (and free of other effects too, of course).

A similar challenge long faced *lazy* functional languages at the source level: how could one obtain the power of side-effecting operations without invalidating simple "equational reasoning" based on (Beta) and similar rules? The effective solution discovered in that context is to use *monads* [9, 14]. An obvious idea, therefore, is to use monads in an internal representation (IR) for compilers of call-by-value languages. Some initial steps in this direction were recently taken by Peyton Jones, Launchbury, Shields, and Tolmach [13]. The aim of that work was to design an IR suitable for both eager and lazy source languages. In this paper we pursue the use of monads with particular reference to eager languages (only), and address the question of how to discover and record several *different sorts* of effects in a single, unified monadic type system. We introduce a *hierarchy* of monads, ordered by increasing "strength of effect," and an inference algorithm for annotating source program subexpressions with their minimal effect.

Past approaches to coping with effects have fallen into two main camps. One approach (used, e.g., by SML of New Jersey [1] and the TIL compiler [17]) is to fall back on a weaker form of (Beta), called (Beta$_v$), which *is* valid in eager settings. (Beta$_v$) restricts the bound expression e to variables, constants, and λ-abstractions; since "evaluating" these expressions never actually causes any computation, they can be moved and substituted with impunity. To augment this rule, these compilers use local syntactic analysis to discover expressions that are demonstrably pure and terminating. Local syntactic analysis must assume that calls to unknown functions may be impure and non-terminating. Still, this form of analysis can be quite effective, particularly if the compiler inlines functions enthusiastically. The other approach (used, e.g., by the ML Kit compiler [4]) uses a sophisticated *effect inference* system [15] to track the latent effects of functions on a very detailed basis. The goals of this school are typically more far-reaching; the aim is to use effects information to provide more generous

[1] Of course, the fact that a transformation is valid doesn't mean that applying it will necessarily improve the program. For example, (Beta) (oriented left-to-right) is not an improving transformation if e_1 is expensive to compute and x appears many times in e_2; similarly, (RecHoist) (oriented left-to-right) is not improving if f is not applied in e_3.

polymorphic generalization rules (e.g., as in [21, 16]), or to perform significantly more sophisticated optimizations, such as automatic parallelization [6] or stack-allocation of heap-like data [18]. In support of these goals, effect inference has generally been used to track store effects at a fine-grained level.

Our approach is essentially a simple monomorphic variant of effect inference applied to a wider variety of effects (including non-termination, exceptions, and IO), cast in monadic form, and intended to support transformational code-motion optimizations. We infer information about latent effects, but we do not attempt to calculate effects at a very fine level of granularity. In return, our inference system is particularly simple to state and implement. However, there is nothing fundamentally new about our system as compared with that of Talpin and Jouvelot [15], except our decision to use a monadic syntax and validate it using a typed monadic semantics. A practical advantage of the monadic syntax is that it makes it easy to reflect the results of the effect inference in the program itself, where they can be easily consulted (and kept up to date) by subsequent optimizations, rather than in an auxiliary data structure. An advantage of the monadic semantics is that it provides a natural foundation for probing and proving the correctness of transformations in the presence of a variety of effects.

In related work, Wadler [20] has recently and independently shown that Talpin and Jouvelot's effect inference system can be applied in a monadic framework; he uses an untyped semantics, and considers only store effects. In another independent project, Benton and Kennedy are prototyping an ML compiler with an IR that describes effects using a monadic encoding similar to ours [3].

2 Source Language

This section briefly describes an ML-like source language we use to explain our approach. The call-by-value source language is presented in Fig. 1. It is a simple, monomorphic variant of ML, expressed in A-normal form [5], which names the result of each computation and makes evaluation order completely explicit. The class `const` includes primitive functions as well as constants. The `Let` construct is monomorphic; that is, `Let(`x`,`e_1`,`e_2`)` has the same semantics and typing properties as would `App(Abs(`x`,`e_2`),`e_1`)` (were this legal A-normal form). The restriction to a monomorphic language is not essential (see Sect. 5). All functions are unary; primitives like `Plus` take a two-element tuple as argument. For simplicity of presentation, we restrict `Letrec` to single functions.

The types of constants are given in Fig. 2. Exceptions carry values of type `Exn`, which are nullary exception constructors. `Raise` takes an exception constructor; rather than providing a means for declaring such constructors, we assume an arbitrary pool of constructor constants. `Handle` catches *all* exceptions that are raised while evaluating its first argument and passes the associated exception value to its second argument, which must be a handler function expecting an `Exn`. The body of the handler function may or may not choose to reraise the exception depending on its value, which may be tested using `EqExn`.

```
datatype typ =
  Int
| Bool
| Exn
| Tup of typ list
| -> of typ * typ

datatype const =
  Integer of int
| True | False
| DivByZero | ...
| Plus | Minus | Times
| Divide
| EqInt | LtInt
| EqBool | EqExn
| WriteInt
| ...
```

```
type varty = var * typ

datatype value =
  Var of var
| Const of const

datatype exp =
  Val of value
| Abs of varty * exp
| App of value * value
| If of value * exp * exp
| Let of varty * exp * exp
| Letrec of varty * varty * exp * exp
| Tuple of value list
| Project of int * value
| Raise of value
| Handle of exp * value
```

Fig. 1. Abstract syntax for source language (presented as ML datatype)

```
                 Integer _ : Int
               True,False : Bool
                DivByZero : Exn
Plus,Minus,Times,Divide : Tup[Int,Int] -> Int
              EqInt,LtInt : Tup[Int,Int] -> Bool
                   EqBool : Tup[Bool,Bool] -> Bool
                    EqExn : Tup[Exn,Exn] -> Bool
                 WriteInt : Int -> Tup[]
```

Fig. 2. Typings for constants in initial environment

.

The primitive function `Divide` has the potential to raise a particular exception `DivByZero`. We supply `WriteInt` as a paradigmatic state-altering primitive; internal side-effects such as ML reference manipulations would be handled similarly. All other primitives are pure and guaranteed to terminate. The semantics of the remainder of the language are completely ordinary.

3 Intermediate Representation with Monadic Types

Figure 3 shows the abstract syntax of our monadic intermediate representation (IR). (For an example of the code, look ahead to Fig. 11.) For the most part, terms are the same as in the source language, but with the addition of monad annotations on `Let` and `Handle` constructs and a new `Up` construct; these are described in detail below.

```
datatype monad = ID | LIFT | EXN | ST

datatype mtyp = M of monad * vtyp
and vtyp =
  Int
| Bool
| Exn
| Tup of vtyp list
| -> of vtyp * mtyp

type varty = var * vtyp

datatype value =
  Var of var
| Const of const

datatype exp =
  Val of value
| Abs of varty * exp
| App of value * value
| If of value * exp * exp
| Let of monad * monad * varty * exp * exp
| Letrec of varty * varty * exp * exp
| Tuple of value list
| Project of int * value
| Raise of mtyp * value
| Handle of monad * exp * value
| Up of monad * monad * exp
```

Fig. 3. Abstract syntax for monadic typed intermediate representation

```
          Integer _ : Int
         True,False : Bool
          DivByZero : Exn
 Plus,Minus,Times : Tup[Int,Int] -> M(ID,Int)
             Divide : Tup[Int,Int] -> M(EXN,Int)
       EqInt,LtInt : Tup[Int,Int] -> M(ID,Bool)
             EqBool : Tup[Bool,Bool] -> M(ID,Bool)
              EqExn : Tup[Exn,Exn] -> M(ID,Bool)
           WriteInt : Int -> M(ST,Tup[])
```

Fig. 4. Monadic typings for constants in initial environment

Values have ordinary value types (`vtyps`); expressions have monadic types (`mtyps`), which incorporate a `vtyp` and a monad (possibly the identity monad, `ID`). Since this is a call-by-value language, the domain of each arrow types is a `vtyp`, but the codomain is an arbitrary `mtyp`. The monadic types for the constants are specified in Fig. 4. The typing rules are given in Fig. 5. In this figure, and throughout our discussion, t ranges value types, m over monads, v over values, c over constants, x,y,z,f over variables, and e over expressions.

For this presentation, we use four monads arranged in a simple linear order. In order of "increasing effect," these are:

- `ID`, the identity monad, which describes pure, terminating computations.
- `LIFT`, the lifting monad, which describes pure but potentially non-terminating computations.
- `EXN`, the monad of exceptions and lifting, which describes computations that may raise an (uncaught) exception, and are potentially non-terminating.
- `ST`, the monad of state, exceptions, and lifting, which describes computations that may write to the "outside world," may raise an exception, and are potentially non-terminating.

We write $m_1 < m_2$ iff m_1 precedes m_2 on this list. Intuitively, $m_1 < m_2$ implies that computations in m_2 are "more effectful" than those in m_1; they can provoke any of the effects in m_1 and then some. This particular hierarchy captures a number of distinctions that are useful for transforming ML programs. We discuss the extension of our approach to more elaborately stratified monadic structures in Sect. 6.

More formally, suppose for each monad m we are given the standard operations $unit_m$, which turns values into null computations in m, and $bind_m$, which composes computations in m, and that the usual monad laws hold:

(Left) $$bind_m\ (unit_m x)\ k = k\ x$$

(Right) $$bind_m\ e\ unit_m = e$$

(Assoc) $$bind_m\ e\ (\lambda x.bind_m\ (k\ x)\ h) = bind_m(bind_m\ e\ k)\ h$$

Moreover, suppose that for each value type t and monad m, $\mathcal{M}[\![m]\!](\mathcal{T}[\![t]\!])$ gives the domain of values of type $\mathtt{M(}m\mathtt{,}t\mathtt{)}$. Then $m_1 < m_2$ implies that there exists an unique embedding $up_{m_1 \to m_2}$ which, for every value type t, maps $\mathcal{M}[\![m_1]\!](\mathcal{T}[\![t]\!])$ to $\mathcal{M}[\![m_2]\!](\mathcal{T}[\![t]\!])$. The up functions, sometimes called monad morphisms or lifting functions [10], obey these laws:

(Unit) $$up_{m_1 \to m_2} \circ unit_{m_1} = unit_{m_2}$$

(Bind) $$up_{m_1 \to m_2}(bind_{m_1}\ e\ k) = bind_{m_2}(up_{m_1 \to m_2}\ e)\ (up_{m_1 \to m_2} \circ k)$$

The up functions can also be viewed as generalizations of $unit$ operations, since, by (Unit), $up_{\mathtt{ID} \to m} = unit_m$. Fig. 6 gives semantic interpretations for types as

$$\frac{E(v) = t}{E \vdash_v \texttt{Var}\ \ v : t}$$

$$\frac{\texttt{Typeof(}c\texttt{)}\ \texttt{=}\ t}{E \vdash_v \texttt{Const}\ \ c : t}$$

$$\frac{E \vdash_v v : t}{E \vdash \texttt{Val}\ \ v : \texttt{M(ID,}t\texttt{)}}$$

$$\frac{E + \{x : t_1\} \vdash e : \texttt{M(}m_2\texttt{,}t_2\texttt{)}}{E \vdash \texttt{Abs(}x : t_1\texttt{,}e\texttt{)} : \texttt{M(ID,}t_1\ \texttt{->}\ \texttt{M(}m_2\texttt{,}t_2\texttt{))}}$$

$$\frac{E \vdash_v v_1 : t_1\ \texttt{->}\ \texttt{M(}m_2\texttt{,}t_2\texttt{)} \quad E \vdash_v v_2 : t_1}{E \vdash \texttt{App(}v_1\texttt{,}v_2\texttt{)} : \texttt{M(}m_2\texttt{,}t_2\texttt{)}}$$

$$\frac{E \vdash_v v : \texttt{Bool} \quad E \vdash e_1 : \texttt{M(}m\texttt{,}t\texttt{)} \quad E \vdash e_2 : \texttt{M(}m\texttt{,}t\texttt{)}}{E \vdash \texttt{If(}v\texttt{,}e_1\texttt{,}e_2\texttt{)} : \texttt{M(}m\texttt{,}t\texttt{)}}$$

$$\frac{E \vdash e_1 : \texttt{M(}m_1\texttt{,}t_1\texttt{)} \quad E + \{x : t_1\} \vdash e_2 : \texttt{M(}m_2\texttt{,}t_2\texttt{)} \quad (m_1 \leq m_2)}{E \vdash \texttt{Let(}m_1\texttt{,}m_2\texttt{,}x : t_1\texttt{,}e_1\texttt{,}e_2\texttt{)} : \texttt{M(}m_2\texttt{,}t_2\texttt{)}}$$

$$\frac{\begin{array}{c} E + \{f : t_0\ \texttt{->}\ \texttt{M(}m_1\texttt{,}t_1\texttt{)}, x : t_0\} \vdash e_1 : \texttt{M(}m_1\texttt{,}t_1\texttt{)} \\ E + \{f : t_0\ \texttt{->}\ \texttt{M(}m_1\texttt{,}t_1\texttt{)}\} \vdash e_2 : \texttt{M(}m_2\texttt{,}t_2\texttt{)} \end{array} \quad (\texttt{LIFT} \leq m_1)}{E \vdash \texttt{Letrec(}f : t_0\ \texttt{->}\ \texttt{M(}m_1\texttt{,}t_1\texttt{)}\texttt{,}x : t_0\texttt{,}e_1\texttt{,}e_2\texttt{)} : \texttt{M(}m_2\texttt{,}t_2\texttt{)}}$$

$$\frac{E \vdash_v v_1 : t_1 \quad \ldots \quad E \vdash_v v_n : t_n}{E \vdash \texttt{Tuple(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)} : \texttt{M(ID,Tup[}t_1, \ldots, t_n\texttt{])}}$$

$$\frac{E \vdash_v v : \texttt{Tup[}t_1, \ldots, t_n\texttt{]} \quad (1 \leq i \leq n)}{E \vdash \texttt{Project(}i\texttt{,}v\texttt{)} : \texttt{M(ID,}t_i\texttt{)}}$$

$$\frac{E \vdash_v v : \texttt{Exn}}{E \vdash \texttt{Raise(M(EXN,}t\texttt{),}v\texttt{)} : \texttt{M(EXN,}t\texttt{)}}$$

$$\frac{E \vdash e : \texttt{M(}m\texttt{,}t\texttt{)} \quad E \vdash_v v : \texttt{Exn}\ \texttt{->}\ \texttt{M(}m\texttt{,}t\texttt{)} \quad (\texttt{EXN} \leq m)}{E \vdash \texttt{Handle(}m\texttt{,}e\texttt{,}v\texttt{)} : \texttt{M(}m\texttt{,}t\texttt{)}}$$

$$\frac{E \vdash e : \texttt{M(}m_1\texttt{,}t\texttt{)} \quad (m_1 \leq m_2)}{E \vdash \texttt{Up(}m_1\texttt{,}m_2\texttt{,}e\texttt{)} : \texttt{M(}m_2\texttt{,}t\texttt{)}}$$

Fig. 5. Typing rules for intermediate language

complete partial orders ($\mathcal{CPO}$'s), and for our monads, together with the associated *up* and *bind* functions. Note that the following laws hold under these semantics:

$$\text{(Id)} \qquad up_{m \to m} = id$$

$$\text{(Compose)} \qquad up_{m_0 \to m_2} = up_{m_1 \to m_2} \circ up_{m_0 \to m_1} \quad (m_0 \leq m_1 \leq m_2)$$

A typed semantics for terms is given in Figs. 7 and 8. Environments ρ map identifiers to values. This semantics is largely predictable. However, the `Let` construct now serves to make the composition of monadic computations explicit, and the `Up` construct makes monadic coercions explicit. Intuitively,

$$\texttt{Let}(m_1,m_2,(x,t_1),e_1,e_2)$$

evaluates e_1, which has monadic type $\texttt{M}(m_1,t)$, performing any associated effects, binds the resulting value to $x : t_1$, and then evaluates e_2, which has monadic type $\texttt{M}(m_2,t_2)$. Thus, it essentially plays the role of the usual monadic *bind* operation; in particular, if $m_1 = m_2$, the semantic interpretation of the above expression in environment ρ is just

$$bind_{m_1}(\mathcal{E}[\![e_1]\!]\rho)(\lambda y.\mathcal{E}[\![e_2]\!]\rho[x := y])$$

However, our typing rules (Fig. 5) require only that $m_2 \geq m_1$; i.e., e_2 may be in a more effectful monad than e_1 The semantics of a general "mixed-monad" `Let` is

$$bind_{m_2}(up_{m_1 \to m_2}(\mathcal{E}[\![e_1]\!]\rho))(\lambda y.\mathcal{E}[\![e_2]\!]\rho[x := y])$$

The term $\texttt{Let(Up(}m_1,m_2,e_1\texttt{)},m_2,(x,t),e_1,e_2\texttt{)}$ has the same semantics, so the more general form of `Let` is strictly redundant. But this form is useful, because it makes it easier to state (and recognize left-hand sides for) many interesting *transformations* involving `Let` whose validity depends on the monad m_1 rather than on m_2. For example, a "non-monadic" `Let`, for which (Beta) is always valid, is simply one in which $m_1 = \texttt{ID}$. Further examples will be shown in Sect. 4.

The semantics of the "non-proper morphism" $\texttt{Handle}(e,v)$ deserve special attention. Expression e may be in either `EXN` or `ST`, and the meaning of `Handle` depends on which; the `ST` version must manipulate the state component. Note that there are two plausible ways to combine state with exceptions. In the semantics we have given (as in ML), handling an exception does not alter the state, but it would be equally reasonable to revert the state on handle. Incidentally, we don't have to give a semantics when e is in `ID` or `LIFT`, because the typing rule for `Handle` disallows these cases. Of course, such cases might appear in source code; to generate monadic IR for them, e can be coerced into `EXN` with an explicit `Up`, or the `Handle` can be omitted altogether in favor of e, which by its type cannot raise an exception! A `Raise` expression is handled similarly; the typing rules force it into monad `EXN`, so semantics need only be given for that case, but the whole expression may be coerced into `ST` by an explicit `Up` if necessary.

$$
\begin{array}{rcll}
\mathcal{T} : \texttt{vtyp} & \rightarrow & \mathcal{CPO} & \\
\mathcal{T}[\![\texttt{Int}]\!] & = & \mathcal{Z} & \\
\mathcal{T}[\![\texttt{Bool}]\!] & = & \mathcal{Z} & \text{(0 represents false)} \\
\mathcal{T}[\![\texttt{Exn}]\!] & = & \mathcal{Z} & \\
\mathcal{T}[\![\texttt{Tup[}t_1, \ldots, t_n\texttt{]}]\!] & = & \mathcal{T}[\![t_1]\!] \times \ldots \times \mathcal{T}[\![t_n]\!] & (n > 0) \\
\mathcal{T}[\![\texttt{Tup[]}]\!] & = & \mathbf{1} & \\
\mathcal{T}[\![t_1 \texttt{ -> M(}m_2\texttt{,}t_2\texttt{)}]\!] & = & \mathcal{T}[\![t_1]\!] \rightarrow \mathcal{M}[\![m_2]\!](\mathcal{T}[\![t_2]\!]) &
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{M} : \texttt{monad} & \rightarrow & \mathcal{CPO} \rightarrow \mathcal{CPO} \\
\mathcal{M}[\![\texttt{ID}]\!]c & = & c \\
\mathcal{M}[\![\texttt{LIFT}]\!]c & = & c_\perp \\
\mathcal{M}[\![\texttt{EXN}]\!]c & = & (\mathbf{Ok}(c) + \mathbf{Fail}(\mathcal{Z}))_\perp \\
\mathcal{M}[\![\texttt{ST}]\!]c & = & \mathbf{State} \rightarrow ((\mathbf{Ok}(c) + \mathbf{Fail}(\mathcal{Z})) \times \mathbf{State})_\perp
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{bind}_{\texttt{ID}}\ x\ k & = & k\ x & \\
\mathit{bind}_{\texttt{LIFT}}\ x\ k & = & k\ a & \text{if } x = a_\perp \\
 & & \perp & \text{if } x = \perp \\
\mathit{bind}_{\texttt{EXN}}\ x\ k & = & k\ a & \text{if } x = \mathbf{Ok}(a)_\perp \\
 & & \mathbf{Fail}(b)_\perp & \text{if } x = \mathbf{Fail}(b)_\perp \\
 & & \perp & \text{if } x = \perp \\
\mathit{bind}_{\texttt{ST}}\ x\ k\ s & = & k\ a\ s' & \text{if } x\ s = (\mathbf{Ok}(a), s')_\perp \\
 & & (\mathbf{Fail}(b), s')_\perp & \text{if } x\ s = (\mathbf{Fail}(b), s')_\perp \\
 & & \perp & \text{if } x\ s = \perp
\end{array}
$$

$$
\begin{array}{rcll}
\mathit{up}_{m \rightarrow m}\ x & = & x & \\
\mathit{up}_{\texttt{ID} \rightarrow \texttt{LIFT}}\ x & = & x_\perp & \\
\mathit{up}_{\texttt{ID} \rightarrow \texttt{EXN}}\ x & = & \mathbf{Ok}(x)_\perp & \\
\mathit{up}_{\texttt{ID} \rightarrow \texttt{ST}}\ x\ s & = & (\mathbf{Ok}(x), s)_\perp & \\
\mathit{up}_{\texttt{LIFT} \rightarrow \texttt{EXN}}\ x & = & \mathbf{Ok}(a)_\perp & \text{if } x = a_\perp \\
 & & \perp & \text{if } x = \perp \\
\mathit{up}_{\texttt{LIFT} \rightarrow \texttt{ST}}\ x\ s & = & (\mathbf{Ok}(a), s)_\perp & \text{if } x = a_\perp \\
 & & \perp & \text{if } x = \perp \\
\mathit{up}_{\texttt{EXN} \rightarrow \texttt{ST}}\ x\ s & = & (\mathbf{Ok}(a), s)_\perp & \text{if } x = \mathbf{Ok}(a)_\perp \\
 & & (\mathbf{Fail}(b), s)_\perp & \text{if } x = \mathbf{Fail}(b)_\perp \\
 & & \perp & \text{if } x = \perp
\end{array}
$$

Fig. 6. Semantics of types and monads

$$
\begin{array}{rcll}
\mathcal{V} : (\texttt{value} : t) \rightarrow Env & \rightarrow & \mathcal{T}[\![t]\!] & \\
\mathcal{V}[\![\texttt{Var}\ v]\!]\rho & = & \rho(v) & \\
\mathcal{V}[\![\texttt{Const (Integer } i\texttt{)}]\!]\rho & = & i & \\
\mathcal{V}[\![\texttt{Const True}]\!]\rho & = & 1 & \\
\mathcal{V}[\![\texttt{Const False}]\!]\rho & = & 0 & \\
\mathcal{V}[\![\texttt{Const Plus}]\!]\rho & = & plus & \\
\ldots\mathcal{V}[\![\texttt{Const Divide}]\!]\rho & = & divideby & \\
\ldots\mathcal{V}[\![\texttt{Const WriteInt}]\!]\rho & = & writeint & \\
\mathcal{V}[\![\texttt{Const DivByZero}]\!]\rho & = & divby0 & \\
\ldots & & & \\
plus\ (a_1, a_2) & = & a_1 + a_2 & \\
divideby\ (a_1, a_2) & = & \mathbf{Ok}(a_1/a_2)_\perp & \text{if } a_2 \neq 0 \\
 & & \mathbf{Fail}(divby0)_\perp & \text{if } a_2 = 0 \\
\mathbf{State} & = & [\mathcal{Z}] & \text{(sequence written out so far)} \\
writeint\ a\ s & = & (\mathbf{Ok}(), append(s, [a]))_\perp & \\
divby0 & = & 42 & \text{(arbitrary fixed integer)}
\end{array}
$$

Fig. 7. Semantics of values

$$
\begin{array}{rcll}
\mathcal{E} : (\texttt{exp} : \texttt{M(}m\texttt{,}t\texttt{)}) \rightarrow Env & \rightarrow & \mathcal{M}[\![m]\!](\mathcal{T}[\![t]\!]) & \\
\mathcal{E}[\![\texttt{Val}\ v]\!]\rho & = & \mathcal{V}[\![v]\!]\rho & \\
\mathcal{E}[\![\texttt{Abs(}x\texttt{,}e\texttt{)}]\!]\rho & = & \lambda y.\mathcal{E}[\![e]\!]\rho[x := y] & \\
\mathcal{E}[\![\texttt{App(}v_1\texttt{,}v_2\texttt{)}]\!]\rho & = & (\mathcal{V}[\![v_1]\!]\rho)\ (\mathcal{V}[\![v_2]\!]\rho) & \\
\mathcal{E}[\![\texttt{If(}v\texttt{,}e_1\texttt{,}e_2\texttt{)}]\!]\rho & = & if\,(\mathcal{V}[\![v]\!]\rho)\ (\mathcal{E}[\![e_1]\!]\rho)\ (\mathcal{E}[\![e_2]\!]\rho) & \\
\mathcal{E}[\![\texttt{Letrec(}f\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{)}]\!]\rho & = & \mathcal{E}[\![e_2]\!](\rho[f := fix(\lambda f'.\lambda v.\mathcal{E}[\![e_1]\!](\rho[f := f', x := v]))]) & \\
\mathcal{E}[\![\texttt{Tuple(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{)}]\!]\rho & = & (\mathcal{V}[\![v_1]\!]\rho, \ldots, \mathcal{V}[\![v_n]\!]\rho) & \\
\mathcal{E}[\![\texttt{Project(}i\texttt{,}v\texttt{)}]\!]\rho & = & proj_i(\mathcal{V}[\![v]\!]\rho) & \\
\mathcal{E}[\![\texttt{Raise(M(EXN,}t\texttt{),}v\texttt{)}]\!]\rho & = & (\mathbf{Fail}(\mathcal{V}[\![v]\!]\rho))_\perp & \\
\mathcal{E}[\![\texttt{Handle(}m\texttt{,}e\texttt{,}v\texttt{)}]\!]\rho & = & handle_m(\mathcal{E}[\![e]\!]\rho)(\mathcal{V}[\![v]\!]\rho) & \\
\mathcal{E}[\![\texttt{Let(}m_1\texttt{,}m_2\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{)}]\!]\rho & = & bind_{m_2}(up_{m_1 \rightarrow m_2}(\mathcal{E}[\![e_1]\!]\rho))(\lambda y.\mathcal{E}[\![e_2]\!]\rho[x := y]) & \\
\mathcal{E}[\![\texttt{Up(}m_1\texttt{,}m_2\texttt{,}e\texttt{)}]\!]\rho & = & up_{m_1 \rightarrow m_2}(\mathcal{E}[\![e]\!]\rho) & \\
 & & & \\
if\ v\ a_t\ a_f & = & a_t & \text{if } v \neq 0 \\
 & & a_f & \text{if } v = 0 \\
proj_i(v_1, \ldots, v_n) & = & v_i & \\
handle_{\texttt{EXN}}\ x\ h & = & \mathbf{Ok}(a)_\perp & \text{if } x = \mathbf{Ok}(a)_\perp \\
 & & h\ a & \text{if } x = \mathbf{Fail}(a)_\perp \\
 & & \perp & \text{if } x = \perp \\
handle_{\texttt{ST}}\ x\ h\ s & = & (\mathbf{Ok}(a), s')_\perp & \text{if } x\ s = (\mathbf{Ok}(a), s')_\perp \\
 & & h\ a\ s' & \text{if } x\ s = (\mathbf{Fail}(a), s')_\perp \\
 & & \perp & \text{if } x\ s = \perp
\end{array}
$$

Fig. 8. Semantics of expressions

(LetLeft) $\quad \texttt{Let(}m_2\texttt{,}m_3\texttt{,}x\texttt{,Up(}m_1\texttt{,}m_2\texttt{,}e_1\texttt{),}e_2\texttt{)} = \texttt{Let(}m_1\texttt{,}m_3\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{)}$
$(m_1 \leq m_2 \leq m_3)$

(LetRight) $\quad \texttt{Let(}m_1\texttt{,}m_2\texttt{,}x\texttt{,}e\texttt{,Up(ID,}m_2\texttt{,Val(Var }x\texttt{)))} = \texttt{Up(}m_1\texttt{,}m_2\texttt{,}e\texttt{)}$
$(m_1 \leq m_2)$

(LetAssoc) $\quad \texttt{Let(}m_2\texttt{,}m_3\texttt{,}x\texttt{,Let(}m_1\texttt{,}m_2\texttt{,}y\texttt{,}e_1\texttt{,}e_2\texttt{),}e_3\texttt{)} =$
$\texttt{Let(}m_1\texttt{,}m_3\texttt{,}y\texttt{,}e_1\texttt{,Let(}m_2\texttt{,}m_3\texttt{,}x\texttt{,}e_2\texttt{,}e_3\texttt{))}$
$(m_1 \leq m_2 \leq m_3; y \notin FV(e_3))$

(IdentUp) $\quad \texttt{Up(}m\texttt{,}m\texttt{,}e\texttt{)} = e$

(ComposeUp) $\quad \texttt{Up(}m_1\texttt{,}m_3\texttt{,}e\texttt{)} = \texttt{Up(}m_2\texttt{,}m_3\texttt{,(Up(}m_1\texttt{,}m_2\texttt{,}e\texttt{)))}$
$(m_1 \leq m_2 \leq m_3)$

(LetUp) $\quad \texttt{Up(}m_2\texttt{,}m_4\texttt{,Let(}m_1\texttt{,}m_2\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{))} =$
$\texttt{Let(}m_3\texttt{,}m_4\texttt{,}x\texttt{,Up(}m_1\texttt{,}m_3\texttt{,}e_1\texttt{),Up(}m_2\texttt{,}m_4\texttt{,}e_2\texttt{))}$
$(m_1 \leq m_2, m_3 \leq m_4)$

Fig. 9. Generalized monad laws

4 Transformation Rules

In this section we attempt to motivate our IR, and in particular our choice of monads, by presenting a number of useful transformation laws. These laws can can be proved correct with respect to the denotational semantics of Sect. 3. The proofs are straightforward but tedious, so are omitted here. Of course, this is by no means a complete set of rules needed by an optimizer; there are many others, both general-purpose and specific to particular operators. Also, as noted earlier, not all valid transformations are improvements.

Figure 9 gives general rules for manipulating monadic expressions. (LetLeft), (LetRight), and (LetAssoc) are generalizations of the usual (Left), (Right), and (Assoc) laws for a single monad, which can be recovered from these rules by setting $m_1 = \texttt{ID}$ and $m_2 = m_3$ in (LetLeft), setting $m_1 = m_2$ in (LetRight), and setting $m_1 = m_2 = m_3$ in (LetAssoc). (IdentUp) and (ComposeUp) are just the (Ident) and (Compose) laws stated in IR syntax; they let us do housekeeping on coercions. Law (Unit) is the special case of (ComposeUp) obtained by setting $m_1 = \texttt{ID}$. (LetUp) permits us to move expressions with suitably weak effects in and out of coercions; (Bind) is the special case of (LetUp) obtained by setting $m_1 = m_2$ and $m_3 = m_4$, All these laws have variants involving `Letrec`, in which $\texttt{Letrec(}f\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{):M(}m\texttt{,}t\texttt{)}$ behaves just like $\texttt{Let(ID,}m\texttt{,}f\texttt{,Abs(}x\texttt{,}e_1\texttt{),}e_2\texttt{)}$; we omit the details of these.

Figure 10 lists some valid laws for altering execution order. We have full beta reduction for variables bound in the `ID` monad (BetaID). In general, the order of two bindings can be exchanged if there is no data dependence between them, *and* if either of them is in the `ID` monad (ExchangeID) or both are in or below the `LIFT` monad (ExchangeLIFT). The intuition for the latter rule is that it harmless to reorder two expressions even if one or both may not terminate, because we cannot detect which one causes the non-termination. On the other hand, there is no similar rule for the `EXN` monad, because we *can* distinguish different raised exceptions according to the constructor value they carry. This is the principal difference between `LIFT` and `EXN` for the purposes of code motion.

Rule (RecHoistID) states that it always valid to lift a pure expression out of a `Letrec` (if no data dependence is violated). (RecHoistEXN) reflects a much stronger property: it is valid to lift a non-terminating or exception-raising expression of a `Letrec` *if* the recursive function is guaranteed to be executed at least once. This is the principal advantage of distinguishing `EXN` from the more general `ST` monad, for which the transform is not valid. Although the left-hand side of (RecHoistEXN) may seem a crude way to characterize functions guaranteed to be called at least once, and unlikely to appear in practice, it arises naturally if we systematically introduce loop headers for recursions [2], according to the following law:

$$\text{(Hdr)}\quad \begin{array}{l} \texttt{Letrec(}f\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{):M(}m\texttt{,}t\texttt{)} \;=\\ \texttt{Let(ID,}m\texttt{,}f\texttt{,Abs(}z\texttt{,Letrec(}f'\texttt{,}x\texttt{,}e_1[f'/f]\texttt{,App(}f'\texttt{,}z\texttt{))),}e_2\texttt{)}\\ \hfill (f' \notin FV(e_1); f' \neq z) \end{array}$$

(HandleHoistExn) says that an expression that cannot raise an exception can always be hoisted out of a `Handle`. Finally, (IfHoistID), (ThenHoistID), and (AbsHoistID) show the flexibility with which `ID` expressions can be manipulated; these are more likely to be useful when oriented right-to-left ("hoisting down" into conditionally executed code). As before, all these rules have variants involving `Letrec` in place of `Let(ID,...)`, which we omit here.

As a (rather artificial) example of the power of these transformations, consider the code in Fig. 11. The computation of `w` is invariant, so we would like to hoist it above recursive function `r`. Because the binding for `w` is marked as pure and terminating, it can be lifted out of the `if` using (IfHoistID), and can then be exchanged with the pure bindings for `s` and `t` using (ExchangeID). This positions it to be lifted out of `r` using (RecHoistID). Note that the monad annotations tell us that `w` is pure and terminating even though it invokes the unknown function `g`, which is actually bound to `h`.

The example also exposes the limitations of monomorphic effects: if `f` were also applied to an impure function, then `g` and hence `w` would be marked as impure, and the binding for `w` would not be hoistable. In practice, it might be desirable to clone separate copies of `f`, specialized according to the effectfulness of their `g` argument. Worse yet, consider a function that is naturally parametric in its effect, such as `map`. Such a function will always be pessimistically annotated with an effect reflecting the most-effectful function passed to it within the program. The obvious solution is to give functions like `map` a generic type abstracted over a monad variable, analogous to an effect variable in the system of Talpin and Jouvelot [15]. We believe our system can be extended to handle such generic types, but we have not examined the semantic issues involved in detail.

$$\text{(BetaID)} \qquad \texttt{Let(ID,}m\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{)} = e_2[e_1/x]$$

$$\text{(ExchangeID)} \qquad \begin{array}{l} \texttt{Let(}m_1\texttt{,}m_3\texttt{,}x_1\texttt{,}e_1\texttt{,Let(}m_2\texttt{,}m_3\texttt{,}x_2\texttt{,}e_2\texttt{,}e_3\texttt{))} = \\ \texttt{Let(}m_2\texttt{,}m_3\texttt{,}x_2\texttt{,}e_2\texttt{,Let(}m_1\texttt{,}m_3\texttt{,}x_1\texttt{,}e_1\texttt{,}e_3\texttt{))} \\ (m_1 = \texttt{ID}or\ m_2 = \texttt{ID}; x_1 \notin FV(e_2); x_2 \notin FV(e_1)) \end{array}$$

$$\text{(ExchangeLIFT)} \qquad \begin{array}{l} \texttt{Let(}m_1\texttt{,}m_3\texttt{,}x_1\texttt{,}e_1\texttt{,Let(}m_2\texttt{,}m_3\texttt{,}x_2\texttt{,}e_2\texttt{,}e_3\texttt{))} = \\ \texttt{Let(}m_2\texttt{,}m_3\texttt{,}x_2\texttt{,}e_2\texttt{,Let(}m_1\texttt{,}m_3\texttt{,}x_1\texttt{,}e_1\texttt{,}e_3\texttt{))} \\ \qquad (m_1, m_2 \leq \texttt{LIFT}; x_1 \notin FV(e_2); x_2 \notin FV(e_1)) \end{array}$$

$$\text{(RecHoistID)} \qquad \begin{array}{l} \texttt{Letrec(}f\texttt{,}x\texttt{,Let(ID,}m_2\texttt{,}y\texttt{,}e_1\texttt{,}e_2\texttt{),}e_3\texttt{):M(}m_3\texttt{,}t\texttt{)} = \\ \texttt{Let(ID,}m_3\texttt{,}y\texttt{,}e_1\texttt{,Letrec(}f\texttt{,}x\texttt{,}e_2\texttt{,}e_3\texttt{))} \\ \qquad\qquad (f, x \notin FV(e_1); y \notin FV(e_3)) \end{array}$$

$$\text{(RecHoistEXN)} \qquad \begin{array}{l} \texttt{Letrec(}f\texttt{,}x\texttt{,Let(}m_1\texttt{,}m_2\texttt{,}y\texttt{,}e_1\texttt{,}e_2\texttt{),App(}f\texttt{,}v\texttt{))} = \\ \texttt{Let(}m_1\texttt{,}m_2\texttt{,}y\texttt{,}e_1\texttt{,Letrec(}f\texttt{,}x\texttt{,}e_2\texttt{,App(}f\texttt{,}v\texttt{)))} \\ \qquad\qquad (m_1 \leq \texttt{EXN}; f, x \notin FV(e_1); y \neq v) \end{array}$$

$$\text{(HandleHoistEXN)} \qquad \begin{array}{l} \texttt{Handle(}m_2\texttt{,Let(}m_1\texttt{,}m_2\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{),}v\texttt{)} = \\ \texttt{Let(}m_1\texttt{,}m_2\texttt{,}x\texttt{,}e_1\texttt{,Handle(}m_2\texttt{,}e_2\texttt{,}v\texttt{))} \\ \qquad\qquad (m_1 \leq \texttt{EXN}; x \neq v) \end{array}$$

$$\text{(IfHoistID)} \qquad \begin{array}{l} \texttt{If(}v\texttt{,Let(ID,}m\texttt{,}x\texttt{,}e_1\texttt{,}e_2\texttt{),}e_3\texttt{)} = \\ \texttt{Let(ID,}m\texttt{,}x\texttt{,}e_1\texttt{,If(}v\texttt{,}e_2\texttt{,}e_3\texttt{))} \\ \qquad\qquad (x \notin FV(e_3); x \neq v) \end{array}$$

$$\text{(ThenHoistID)} \qquad \begin{array}{l} \texttt{If(}v\texttt{,}e_1\texttt{,Let(ID,}m\texttt{,}x\texttt{,}e_2\texttt{,}e_3\texttt{))} = \\ \texttt{Let(ID,}m\texttt{,}x\texttt{,}e_2\texttt{,If(}v\texttt{,}e_1\texttt{,}e_3\texttt{))} \\ \qquad\qquad (x \notin FV(e_1); x \neq v) \end{array}$$

$$\text{(AbsHoistID)} \qquad \begin{array}{l} \texttt{Abs(}x : t\texttt{,Let(ID,}m\texttt{,}y\texttt{,}e_1\texttt{,}e_2\texttt{))} = \\ \texttt{Let(ID,ID,}y\texttt{,}e_1\texttt{,Abs(}x : t\texttt{,}e_2\texttt{))} \\ \qquad\qquad (x \notin FV(e_1); y \neq x) \end{array}$$

Fig. 10. Code motion laws for monadic expressions

```
let f:(Int -> M(ID,Int * Int)) -> M(ST,Int) =
     fn (g:Int->M(ID,Int * Int)) =>
         letrec r (x:Int) : M(ST,Int) =
           letID t:Int * Int = (x,1)
           in letID s:Bool = EqInt(t)
              in if s then
                   Up(ID,ST,0)
                 else
                   letID w:Int * Int = g(3)
                   in letID y:Int = Plus(w)
                      in letID z:Int * int = (x,y)
                         in letEXN x':Int = Divide(z)
                            in letST dummy:() = WriteInt(x')
                               in r(x')
         in r(10)
in let h:Int->M(ID,Int * Int) = fn (p:Int) => (p,p)
   in f(h)
```

Fig. 11. Example of intermediate code, presented in an obvious concrete analogue of the abstract syntax

5 Monad Inference

It would be possible to translate source programs into type-correct IR programs by simply assuming that *every* expression falls into the maximally-effectful monad (`ST` in our case). Every source `Let` would become a `LetST`, every variable and constant would be coerced into `ST`, and every primitive would return a value in `ST`. Peyton Jones *et al.* [13] suggest performing such a translation, and then using the monad laws (analogous to those in Fig. 9) and the worker-wrapper transform [12] to simplify the result, hopefully resulting in some less-effectful expression bindings. The main objection to this approach is that it doesn't allow calls to unknown functions (for which worker-wrapper doesn't apply) to return non-`ST` results. For example, in the code of Fig. 11, no local syntactic analysis could discover that argument function `g` is pure and terminating.

To obtain better control over effects, we have developed an inference algorithm for computing the minimal monadic effect of each subexpression in a program. Pure, provably terminating expressions are placed in `ID`, pure but potentially non-terminating expressions in `LIFT`, and so forth. The algorithm deals with the latent monadic effects in functions, by recording them in the result types. As an example, it produces the annotations shown in Fig. 11.

The input to the algorithm is an typed program in the source language; the output is a program in the monadically typed IR. The term translation is essentially trivial, since the source and target have identical term structure, except for the possible need for `Up` terms in the target. Consider, for example, the source term `If(`x`,Val `y`,Raise `z`)`. Since `Val` y is a value, its translation is in the `ID` monad, whereas the translation of `Raise` z must be in the `EXN` or `ST`

$$\frac{E \vdash_v v : \texttt{Bool} \quad E \vdash e_1 \Rightarrow e_1' : \texttt{M}(m_1,t) \quad E \vdash e_2 \Rightarrow e_2' : \texttt{M}(m_1,t) \quad (m_1 \leq m_2)}{E \vdash \texttt{If}(v,e_1,e_2){:}t \Rightarrow \texttt{Up}(m_1,m_2,\texttt{If}(v,e_1',e_2')) : \texttt{M}(m_2,t)}$$

$$\frac{E \vdash e_1 \Rightarrow e_1' : \texttt{M}(m_1,t_1) \quad E + \{x : t_1\} \vdash e_2 \Rightarrow e_2' : \texttt{M}(m_2,t_2) \quad (m_1 \leq m_2 \leq m_3)}{E \vdash \texttt{Let}(x : t_1,e_1,e_2){:}t_2 \Rightarrow \texttt{Up}(m_2,m_3,\texttt{Let}(m_1,m_2,x : t_1,e_1',e_2')) : \texttt{M}(m_3,t_2)}$$

$$\frac{E \vdash_v v : \texttt{Exn} \quad (\texttt{EXN} \leq m)}{E \vdash \texttt{Raise}(t,v){:}t \Rightarrow \texttt{Up}(\texttt{EXN},m,\texttt{Raise}(\texttt{M}(\texttt{EXN},t),v)) : \texttt{M}(m,t)}$$

Fig. 12. Selected translation rules

monad. To glue together these subterm translations we must insert a coercion around the translation of the `Val` term. `Up` terms serve exactly this purpose; they add the necessary flexibility to the system to permit all monad constraints to be met. Such a coercion is potentially needed around each subterm in the program.

To develop a deterministic, syntax-directed, translation, we turn each typing rule in Fig. 5 (*except* `Up`) into a translation rule, simply by recording the inferred type and monad information in the appropriate annotation slots of the output, combining the translations of subterms in the obvious manner, and wrapping an `Up` term around the result. As examples, Fig. 12 shows the translation rules corresponding to the typing rules for `If`, `Let`, and `Raise`. Each free type and monad in the translated typed term is initially set to a fresh variable; the translation algorithm generates a set of constraints relating these variables just as in an ordinary type inference algorithm. We discuss the solution of these constraints below. As specified here, the translation is profligate in its introduction of `Up` coercion terms, most of which will prove (after constraint resolution) to be unnecessary identity coercions. We use a postprocessing step to remove unneeded coercions using the (IdentUp) rule.

The translation algorithm generates constraints between types and between monads. Type constraints can be solved using ordinary unification, except that unifying the codomain `mtyp`s of two arrow types requires that their monad components be equated as well as their `vtyp` components. The interesting question is how to record and resolve constraints on the monad variables. Such constraints are introduced explicitly by the side conditions in the `Let`, `Letrec`, and `Up` rules, implicitly by the equating of monads from subexpressions in the `If` and `Handle` rules, and (even more) implicitly as a result of ordinary unification of arrow types, which mention monads in their codomains. The side-condition constraints are all inequalities of the form $m_1 \geq m_2$, where m_1 is a monad variable and m_2 is a variable or an explicit monad. The implicit constraints are all equalities $m_1 = m_2$; for uniformity, we replace these by a pair of inequalities: $m_1 \geq m_2$ and $m_2 \geq m_1$. We collect constraints as a side-effect of the translation process, simply by adding them to a global list.

It is very common for there to be circularities among the monad constraints. To solve the constraint system, we view it as a directed graph with a node for each monad and monad variable, and an edge from m_1 to m_2 for each constraint $m_1 \geq m_2$. We then partition the graph into its strongly connected components, and sort the components into reverse topological order. We process one component at a time, in this order. Since $\geq$ is anti-symmetric, all the nodes in a given component must be assigned the same monad; once this has been determined, it is assigned to all the variables in the component before proceeding to the next component. To determine the minimum possible correct assignment for a component, we consult all the edges from nodes in that component to nodes *outside* the component; because of the order of processing, these nodes must already have received a monad assignment. The maximum of these assignments is the minimum correct assignment for this component. If there are no such edges, the minimum correct assignment is `ID`. This algorithm is linear in the number of constraints, and hence in the size of the source program.

To summarize, we perform monad inference by first translating the source program into a form padded with coercion operators and annotated with monad variables, meanwhile collecting constraints on these variables, and then solving the resulting constraint system to fill in the variables in the translated program. The resulting program will contain many null coercions of the form `Up(`m`,`m`,`e`)`; these can be removed by a single postprocessing pass.

Our algorithm is very similar to a that of Talpin and Jouvelot [15], restricted to a monomorphic source language. Both algorithms generate essentially the same sets of constraints. Talpin and Jouvelot solve the effect constraints using an extended form of unification rather than by a separate mechanism.

It would be natural to extend our algorithm to handle Hindley-Milner polymorphism for both types and monads in the Talpin-Jouvelot style. The idea is to generalize all free type and effect variables in `let` definitions and allow different uses of the bound identifier to instantiate these in different ways. In particular, parametric functions like `map` could be used with many different monads, without one use "polluting" the others. Functions not wholly parametric in their effects would place a minimum effect bound on permissible instantiations for monad variables. Supporting this form of monad polymorphism seems desirable even if there is no type polymorphism (e.g., because the program has already been explicitly monomorphized [19]).

In whole-program compilation of a monad-polymorphic program, the complete set of effect instantiations for each polymorphic definition would be known. This set could be used to put an upper effect bound on monad variables within the definition body and hence determine what transformations are legal there. Alternatively, it could be used to guide the generation of effect-specific clones as suggested in the previous section. In a separate-compilation setting, monad polymorphism in a library definition would still be useful for client code, but not for the library code: in the absence of complete information about uses of a definition, any variable monad in the body of the definition would need to be treated as `ST`, the most "effectful" monad, for the purposes of performing transformations within the body.

6 Extending the Monad Hierarchy

Our basic approach is not restricted to the linearly-ordered set of monads presented in Sect. 3. It extends naturally to any collection of monads and *up* embedding operations that form a *lattice*, with `ID` as the lattice bottom element. It is clearly reasonable to require a partial order; this is equivalent to requiring that (Ident) and (Compose) hold. From the partial order requirement, the distinguished role for `ID`, and the assumption that each monad obeys (Left), (Right), and (Assoc), and each *up* operation obeys (Unit) and (Bind), we can prove the laws of Fig. 9. (The validity of the laws in Fig. 10 naturally depends on the specific semantics of the monads involved.) By also insisting that any two monads in the collection have a least upper bound under embedding, we guarantee that any two arbitrary expressions (e.g., the two arms of an `if`) can be coerced into a (unique) common monad, and hence that the monad inference mechanism of Sect. 5 will work.

One might be tempted to describe such a lattice by specifying a set of "primitive" monads encapsulating individual effects, and then assuming the existence of arbitrary "union" monads representing combinations of effects. As the `Handle` discussion in Sect. 3 indicates, however, there is often more than one way to combine two effects, so it makes no sense to talk in a general way about the "union" of two monads. Instead, it appears necessary to specify explicitly, for every monad m in the lattice,

- a semantic interpretation for m;
- a definition for $bind_m$;
- a definition of $up_{m \to m'}$ for each $m \leq m'$;[2]
- for each non-proper morphism `NP` introduced in m, a definition of $np_{m'}$ for every $m' \geq m$.

The lack of a generic mechanism for combining monads is rather unfortunate, since it turns the proofs of many transformation laws into lengthy case analyses. We conjecture that restricting attention to *up* operations that represent *natural* monad transformers [10] might help organize such proofs into simpler form.

7 Status and Conclusions

We believe our approach to inferring and recording effects shows promise in its simplicity and its semantic clarity. It remains to be seen whether effects information of the kind described here can be used to improve the performance of ML code in any significant way. To answer this question, we have extended the IR described here to a version that supports full Standard ML; we have implemented the monad inference algorithm for this version, and are currently measuring its effectiveness using the backend of our RML compiler system [19].

[2] Since the (Ident) and (Compose) laws must hold in a partial order, it suffices to define $up_{m \to m'}$ for just enough choices of m, m' to guarantee the existence of least upper bounds, since these definitions will imply the definition for other pairs of monads.

Acknowledgements

We have benefitted from conversations with John Launchbury and Dick Kieburtz, and from exposure to the ideas in their unpublished papers [7, 8]. The comments of the anonymous referees also motivated us to clarify the relationship of our algorithm with the existing work of Talpin and Jouvelot. Phil Wadler made helpful commments on an earlier draft.

References

1. A. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.
2. A. Appel. Loop headers in λ-calculus or CPS. *Lisp and Symbolic Computation*, 7(4):337–343, 1994.
3. N. Benton, July 1997. Personal communication.
4. L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von Neumann machines via region representation inference. In *23rd ACM Symposium on Principles of Programming Languages (POPL'96)*, pages 171–183. ACM Press, 1996.
5. C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. *Proc. SIGPLAN Conference on Programming Language Design and Implementation*, 28(6):237–247, June 1993.
6. D. Gifford, P. Jouvelot, J. Lucassen, and M. Sheldon. FX-87 REFERENCE MANUAL. Technical Report MIT-LCS//MIT/LCS/TR-407, Massachusetts Institute of Technology, Laboratory for Computer Science, Sept. 1987.
7. R. Kieburtz and J. Launchbury. Encapsulated effects. (unpublished manuscript), Oct. 1995.
8. R. Kieburtz and J. Launchbury. Towards algebras of encapsulated effects. (unpublished manuscript), 1997.
9. J. Launchbury and S. Peyton Jones. State in Haskell. *Lisp and Symbolic Computation*, pages 293–351, Dec. 1995.
10. S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, Jan. 1995.
11. S. Peyton Jones. Compiling Haskell by program transformation: A report from the trenches. In *Proceedings of ESOP'96*, volume 1058 of *Lecture Notes in Computer Science*, pages 18–44. Springer Verlag, 1996.
12. S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens. In *Proc. Functional Programming Languages and Computer Architecture (FPCA '91)*, pages 636–666, Sept. 191.
13. S. Peyton Jones, J. Launchbury, M. Shields, and A. Tolmach. Bridging the gulf: a common intermediate language for ml and haskel. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 49–61, San Diego, Jan 1998.
14. S. Peyton Jones and P. Wadler. Imperative functional programming. In *20th ACM Symposium on Principles of Programming Languages (POPL'93)*, pages 71–84, Jan. 1993.
15. J.-P. Talpin and P. Jouvelot. Polymorphic type, region and effect inference. *Journal of Functional Programming*, 2:245–271, 1992.
16. J.-P. Talpin and P. Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, June 1994.

17. D. Tarditi. *Design and Implementation of Code Optimizations for a Type-Directed Compiler for Standard ML.* PhD thesis, Carnegie Mellon University, Dec. 1996. Technical Report CMU-CS-97-108.
18. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1 Feb. 1997.
19. A. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source trans lation. *Journal of Functional Programming*, 1998. (to appear).
20. P. Wadler. The marriage of effects and monads. (unpublished manuscript), Mar. 1998.
21. A. Wright. Typing references by effect inference. In *Proc. 4th European Symposium on Programming (ESOP '92)*, volume 582 of *Lecture Notes in Computer Science*, Feb. 1992.