# Convenient Explicit Effects using Type Inference with Subeffects

**Technical Report MSR-TR-2010-80**

Ross Tate
University of California, San Diego
rtate@cs.ucsd.edu

Daan Leijen
Microsoft Research
daan@microsoft.com

## Abstract

Most programming languages in use today let one freely use arbitrary (side) effects. This is despite the fact that unknown and unrestricted side effects are the cause of many software problems. We propose a programming model where effects are treated in a disciplined way, and where the potential side-effects of a function are apparent in its type signature. In contrast to most effect systems that are meant for internal compiler optimizations, our system is designed to be used by the programmer. Inspired by Haskell, we use a coarse-grained hierarchy of effects, like pure and io, which makes it convenient to read and write type signatures. The type and effect of expressions can also be inferred automatically, and we describe a polymorphic type inference system based on Hindley-Milner style inference.

## 1. Introduction

Procedures in programs are generally quite different from mathematical functions. In particular, they can do more than simply return values: procedures can raise exceptions, fail to terminate, read from or write to a heap, or perform I/O operations. This extra power comes at a cost though. Many useful properties enjoyed by pure mathematical functions are not shared by effectful procedures, so it is harder for programmers to reason about their code and for compilers to perform optimizations.

We propose a programming model where effects are a part of the type signature of a function. Just like types help to structure and clarify code, we believe that effects should be part of the mindset of the programmer. Not only would this enable stronger guarantees and better understanding, it would also encourage a more effect-free style of programming with more optimization opportunities. For example, the squaring function:

$$sqr(x : int) = \{x * x\}$$

Gets the type:

$$sqr : int \xrightarrow{\mathsf{total}} int$$

signifying that $sqr$ has no side effect at all and is a total function from integers to integers. If we add a $print$ statement though:

$$sqr(x : int) = \{print(x); x * x\}$$

the (inferred) type indicates that $sqr$ has an input-output (io) effect:

$$sqr : int \xrightarrow{\mathsf{io}} int$$

Note that there was no need to change the original function nor to promote the expression $x * x$ into the io effect. One of our goals is to make effects convenient for the programmer, so we automatically combine effects together using a hierarchy of subeffects. In particular, this makes it convenient for the programmer to use precise effects without having to insert coercions. For example, we can split Haskell's state monad into three separate effects (read, allocate, and write), while automatically combining these effects when required. Types and effects are inferred automatically using a variant of polymorphic Hindley-Milner-style type inference (Hindley 1969; Milner 1978), and there is a natural subtype relation on effects that will automatically promote an effect when necessary without the need for explicit coercions.

We are somewhat hesitant to call our type system an effect system: many effect systems in the literature are designed to enable internal compiler optimizations (Benton and Buchlovsky 2007) and the effect language is often quite complex. In contrast, our effect system is specifically designed for programmers as part of the surface language. We use a coarse-grained natural hierarchy of effects like pure and io so that the effects are easy to write and understand. Nonetheless, our effects are expressive enough to describe effects in ML and precise enough to recognize common usage patterns of effects. In this paper we make the following contributions:

- We have designed a programming language with explicit, precise, and convenient effects. Furthermore we show that this language has a well-defined unambiguous semantics.
- We show how a coarse-grained hierarchy of effects offers many benefits and insights to the programmer. We precisely describe how different effects are related to each other, and motivate the particular choice of hierarchy on the basis of different desirable properties.
- We show how certain stateful computations can be safely considered pure again (Peyton Jones and Launchbury 1995), and how exception handlers can remove partiality.
- Having to keep track of effects manually would be a large burden: we describe a sound and complete polymorphic type-and-effect-inference system that automatically infers the effect and type of any expression, and automatically promotes effects when necessary.
- We formalize the type system and give type directed translation to monadic System F.

Effect systems have been widely studied from many perspectives, and we draw from much of this prior work. Nevertheless, we present a novel system of effects that builds onto the experience with monads in Haskell, and shows how these various perspectives can be combined into a cohesive system.

## 2. Overview

Before we describe our design, we first take a short look at one of the most prominent programming languages that distinguishes pure expressions from side-effecting ones, namely Haskell. In Haskell, all expressions are by default pure and cannot have a side effect.

The only way to introduce side effects is to use the $IO$ (or $ST$) monad. Take for example the fibonacci function in Haskell:

$$\mathit{fib}\ n = \mathsf{if}\ n \leqslant 0\ \mathsf{then}\ 1\ \mathsf{else}\ \mathit{fib}\ (n-1) + \mathit{fib}\ (n-2)$$

which will be assigned the type $Int \rightarrow Int$ in Haskell which signifies that this a pure function from integers to integers. If we would like to print a message in the base case though, we need to 'lift' everything explicitly into the $IO$ monad as follows:

$$\begin{aligned} \mathit{fib}\ n = \mathsf{if}\ n \leqslant 0\ &\mathsf{then}\ \mathsf{do}\ \{\mathit{print}\ \texttt{"hi"}; \mathsf{return}\ 1\} \\ &\mathsf{else}\ \ \mathsf{do}\ \{x \leftarrow \mathit{fib}\ (n-1); \\ &\qquad\qquad y \leftarrow \mathit{fib}\ (n-2); \\ &\qquad\qquad \mathsf{return}\ (x+y)\} \end{aligned}$$

The type would now become $Int \rightarrow IO\ Int$ signifying that this is a function from $Int$ to an $IO$ monad of $Int$, i.e. a computation that when executed returns and $Int$ and potentially has an input/output effect.

The Haskell solution of using monads to separate pure and side-effecting computations has been quite successful and there exists a multiple of real-world programs that use this. Nevertheless, there is a significant syntactic burden: in order to adapt the original function above to print a message, we had to change the entire function, lifting each sub expression into the $IO$ monad. This is also the reason many Haskell programmers tend to put different effects into one larger monad, as it it quickly becomes tiresome to lift the different kinds of expressions explicitly into the right monad.

More seriously, the Haskell notion of purity is still not be strong enough to enable many interesting transformations. In particular, purity includes the effects of partiality and divergence. For example, an expression can have a type $Int$ but when the value is demanded it might actually diverge or raise an exception. The true type is really not $Int$, but $Int_\perp$ signifying that the value can either be an integer, or be undetermined. Even simple transformations like replacing $0 * x$ with $0$ where $x$ is a variable become invalid under this notion of purity.

### 2.1 Effect types

To address the previous problems, we took a fresh look at programming with monads and side-effects, and developed a small prototype language called $\mathsf{F^{OX}}$ (pronounced 'fox'). In essence one can see it as ML with controlled side effects, or as Haskell with strict semantics and implicit monads. In particular, we use a strict semantics where arguments are evaluated before calling a function. This implies that an expression with type $int$ can really be modeled semantically as an integer (and not as a delayed computation that can potentially diverge or raise an exception).

As a consequence, the *only point where side effects can occur is during function application*. We will write function types as $(\tau_1, ..., \tau_n) \xrightarrow{\epsilon} \tau$ to denote that a function takes arguments of type $\tau_1$ to $\tau_n$, and returns a value of type $\tau$ with a potential side effect $\epsilon$. As apparent from the type, and in contrast to ML, functions need to be fully applied and are not curried. This is to make it immediately apparent where side effects can occur. For example, in a curried language like ML an expression like $f\ x\ y$ can have side effects at different points depending on the arity of the function $f$. In our system this is always explicit as function application is written explicitly, and we have either $f(x, y)$ or $(f(x))(y)$.

### 2.2 The basic effect hierarchy

The potential effects that we support form a partial order under a sub-effect relation $\leqslant$, where we have for example that $\mathsf{pure} \leqslant \mathsf{io}$. In Figure 1 we show a simplified version of the effect hierarchy where each edge $\epsilon_1 \rightarrow \epsilon_2$ implies $\epsilon_1 \leqslant \epsilon_2$. We can roughly divide the effects in three groups, the bottom four pure effects, the middle four state effects, and the top I/O effect.

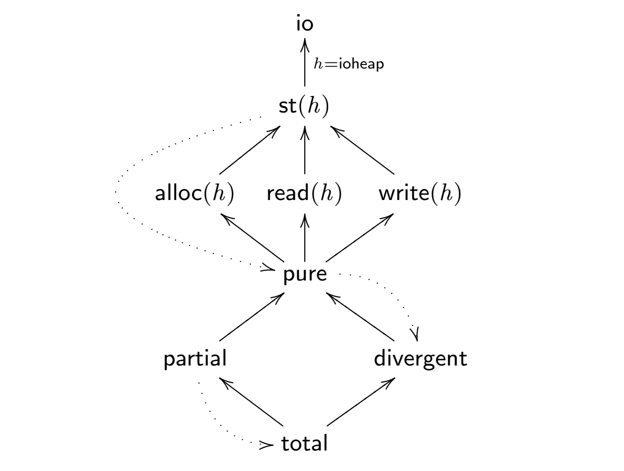

**Figure 1.** The (simplified) effect hierarchy: an arrow $\epsilon_1 \rightarrow \epsilon_2$ implies that $\epsilon_1 \leqslant \epsilon_2$.

$\mathsf{total}$ signifies the absence of any effects. In particular, $\mathsf{total}$ is a bottom element with $\mathsf{total} \leqslant \epsilon$ for any effect $\epsilon$. Functions with a $\mathsf{total}$ effect corresponds basically to total mathematical functions. The effects directly above $\mathsf{total}$ are $\mathsf{partial}$ and $\mathsf{divergent}$, respectively representing expressions that possibly raise an exception or may not terminate. $\mathsf{pure}$ is the join of the those effects. We chose to call this effect *pure* as it corresponds directly to Haskell's notion of purity. The result of functions with the $\mathsf{pure}$ effect is completely determined by the values of the arguments. Indeed, in the absence of arbitrary side effects, such functions behave like mathematical functions and are fully deterministic. The dotted line shows that we can sometimes transform an partial expression to a total one, and a pure expression to a divergent one, when handling exceptions as discussed in Section 3.5.

The middle group of effects all deal with state and are parameterised with a heap argument $h$. Such heap arguments never need to be introduced explicitly by the programmer but will arise naturally as part of type inference. The $\mathsf{alloc}(h)$ effect allocates in the heap $h$, $\mathsf{read}(h)$ reads the heap, and $\mathsf{write}(h)$ writes to the heap. The $\mathsf{st}(h)$ effect is the join of those effects and can potentially allocate, read, and write to the heap. It is useful to distinguish the read, write, and allocate effects as all three have different properties. For example, reads commute with each other and can for example be safely parallelized, as we discuss in Section 3.2. The dotted line implies that we can sometimes safely convert a stateful expression into a pure one as discussed in Section 3.1. The hierarchy of stateful effects is simplified here and well will refine it further in Section 3.1.

Finally, the $\mathsf{io}$ effect represeents what is more traditionally thought of as side effecting and can perform arbitrary I/O operations like file I/O, networking, and generally make any external procedure calls – all bets are off! and unfortunately, for most programming languages one must assume that expressions are always in this particular effect. Note that $\mathsf{io}$ is not a sub effect of arbitrary stateful effects, only $\mathsf{st}(h)$ effects where the heap $h$ is equal to the type constant $\mathsf{ioheap}$. We discuss this in more detail in Section 3.1.

## 3. Programming with effects

Now that the high level hierarchy of effects is known, we will discuss various properties and details of the effect system in the following sections.

### 3.1 State

Using stateful operations, we can for example describe a linear version of the fibonacci function:

```
fib(n) {
  f1 ← newref(1);
  f2 ← newref(1);
  repeat(n − 1){
    sum ← !f1 + !f2;
    f1 := !f2;
    f2 := sum;
  }
  !f2;
}
```

In the above code $newref$ creates a new reference, $f := e$ assigns $e$ to a reference $f$, and $!f$ dereferences a reference. The statement $x \leftarrow e$ binds a name to the result of an expression. The $\mathsf{repeat}(n)$ statement executes its body $n$ times. A valid type for $\mathit{fib}$ is:

$$\mathit{fib} : \forall h.\ int \xrightarrow{\mathsf{st}(h)} int$$

reflecting that the function allocates, reads, and writes into some heap $h$. As apparent, there is no need for the programmer to explicitly manage heaps but the type system introduces such names automatically as part of normal type inference.

***From state to pure*** Interestingly, the above function could be considered pure though: for any input, it always returns the same output since the heap $h$ cannot be modified or observed from outside this function. In particular, we can safely convert any function with an effect $\mathsf{st}(h)$ to a pure function when the heap $h$ is inaccessible from outside. It can be shown that this is exactly the case whenever the function is polymorphic in the heap $h$ and where $h$ is not among the free type variables of argument types or result type. This notion is formalized in the next section and corresponds directly to the use of the higher-ranked $runST$ function in Haskell (Peyton Jones and Launchbury 1995). As we will discuss later, we can generally not apply this rule automatically in the type inferencer and the programmer needs to explicitly insert a run statement as follows:

$$\mathit{fib}(n)\ = \mathsf{run}\{\ldots\}$$

Now, the type inferred can be safely refined to be pure:

$$\mathit{fib} : int \xrightarrow{\mathsf{pure}} int$$

We represent the run transition with the upper dotted arrow in Figure 1.

***A more refined state*** Even the previous type can be further refined. As said before, the diagram in Figure 1 is somewhat simplified. In the full system, there are four variants of each heap effect, one for each of the four basic effects: total, partial, divergent, or pure. For example, the stateful type of the original fibonacci function becomes:

$$\mathit{fib} : \forall h.\ int \xrightarrow{\mathsf{st\text{-}total}(h)} int$$

signifying that this is a stateful, but total function, i.e. it will terminate and raise no exceptions.

The more informative heap effects still form a partial order but the simple arrow between the previous $\mathsf{read}(h)$ and $\mathsf{st}(h)$ effect now becomes a cube as shown in Figure 2, and similarly for the other heap effects. The main advantage distinguishing these heap effects is that we can revert to an even more specific effect when using the run statement. In particular, with the run statement, the type inferred for $\mathit{fib}$ function will actually be total:

$$\mathit{fib} : int \xrightarrow{\mathsf{total}} int$$

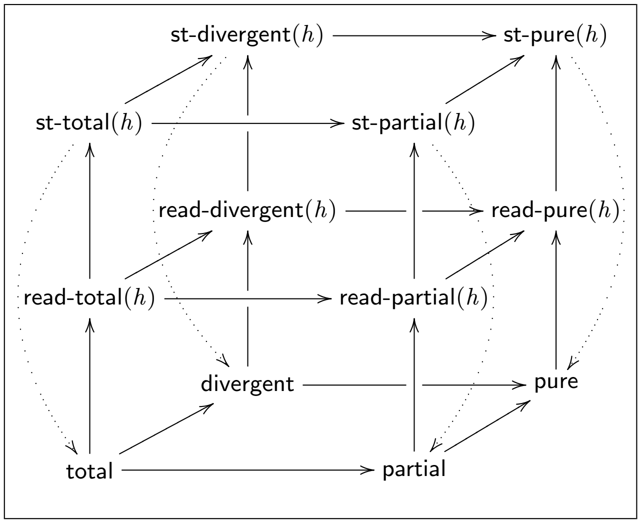

**Figure 2.** The full relation between the basic effects and the expanded read and st effect.

In Figure 2 the valid transitions induced by run are denoted by the dotted downward arrows. It may seem that we could parameterise a heap effect directly with a nested effect, as in $\mathsf{st}(h, \mathsf{total})$, but as shown in Section 6.3 this would lead to an unsound type system and we need to use different effect constants instead.

The example program uses state in a rather limited fashion and uses the heap references just like local variables. Generally though stateful effects can do arbitrary heap manipulation and work across function boundaries. An extended example of the use of isolated state for efficient graph algorithms is described by King and Launchbury (1995).

### 3.2 Commutative effects

Any effects under read-divergent, read-partial, alloc-divergent, or alloc-partial, have the special property that they are *commutative* effects. This means that it is not observable in which order such effects are executed. For example, for any two nullary functions $f_1$ and $f_2$ with a commutative effect and unit result, we have that $f_1(); f_2()$ is equivalent to $f_2(); f_1()$. Note that we cannot include pure in the commutative effects since this can change the nontermination behaviour. With partial, the kind of exception that is raised might differ but we will address this in the way we define exception handlers in Section 3.5. Similarly, allocation might be done in a different order, but we offer no ordering or $addressOf$ operations on references making the difference unobservable.

A common issue in strict programming languages is the order of evaluation of function arguments. Some languages (like C) leave this unspecified but in the presence of side effects this might lead to unpredictable results. For example, the expression $foo(print(\texttt{"a"}), print(\texttt{"b"}))$ might print either "ab" or "ba". This is a innocent example but in general unexpected side effects in arguments can cause bugs that are hard to find. To avoid these issues, we require that all function arguments have commutative effects. The order in which arguments are now evaluated becomes irrelevant since the order is unobservable.

As shown in the previous fibonacci function it is very useful to allow arbitrary commutative effects where we used the addition operator on arguments with a read effect:

$$sum \leftarrow !f_1 + !f_2$$

Explicit monadic programming can be quite cumbersome in these cases as all the the reads have to be both lifted explicitly and performed in an explicit order.

We would still need to order non-commutative arguments explicitly though. For example, our earlier C example must be written as $x \leftarrow print(\texttt{"a"}); y \leftarrow print(\texttt{"b"}); foo(x, y)$ where the order of the print statements is made explicit – a good thing!

### 3.3 Polymorphic effects

As already apparent from the previous examples, the type system supports full parametric polymorphism a la Hindley-Milner (Hindley 1969; Milner 1978). For example, the *head* function can return the head element of any list regardless of the element type:

$$head : \forall\alpha.\ list(\alpha) \xrightarrow{\mathsf{partial}} \alpha$$

This form of polymorphism extends naturally to effects too. Take for example the function *map* that applies a function to all elements of a list:

```
map(f, xs) {
  match (xs){
    cons(x, xx) → { y  ← f(x);
                    yy ← map(f, xx);
                    cons(y, yy) }
    nil         → nil
  }
}
```

which will get the type:

$$map : \forall\alpha\beta\epsilon.\ (\alpha \xrightarrow{\epsilon} \beta, list(\alpha)) \xrightarrow{\epsilon} list(\beta)$$

The inferred type for *map* naturally expresses that the (side) effects of executing *map* is dependent on the effects that the provided function has. For example, $map(print, [\texttt{"a"}, \texttt{"b"}])$ will have an io effect while $map(sqr, [1, 2])$ will be total.

### 3.4 No value restriction

Supporting both Hindley-Milner style polymorphic type inference and effects must be done carefully, since polymorphic type inference is generally unsound in the presence of unrestricted side effects. We illustrate the problem with an example of the ML language. ML also has polymorphic type inference and allows (implicit) side effects. As an example, take the following ML program

```
let r = ref []
in r := [true]; head !r + 1
```

First a reference to an empty list is created. Subsequently, we assign a list with a boolean element to this reference, and then read the element back (*head* $!r$) as an integer! This is clearly unsound and the above program is not a valid ML program. Nevertheless, the program would type check fine under standard Hindley-Milner type inference: the type of *ref* $[\,]$ is generalized and $r$ is assigned the type $\forall\alpha.\ ref(list(\alpha))$ which can be instantiated to both a reference to a list of integers, but also to a reference to a list of booleans. To avoid this issue, the ML language adds the *value restriction* where only expressions that are syntactically a value can be generalized. Therefore, the above code fragment is rejected since $r$ gets a monomorphic type and cannot be used as a reference for both integer lists and boolean lists. Unfortunately, this restriction also rejects many programs that are sound since there is no side effect involved, for example, let $revlists = map\ rev$ is rejected, while let $revlists\ xs = map\ rev\ xs$ is accepted.

When side-effects are known, this issue disappears and there is no need for the value restriction at all. In particular, we can safely generalize over precisely those expressions that have *idempotent* effects, i.e. those effects where executing $\{\, x \leftarrow e; (x, x) \,\}$ is equal to executing $\{\, x \leftarrow e; y \leftarrow e; (x, y) \,\}$. In particular, this holds for any effect under pure, for read and write, but not for alloc, st, or io. For the purposes of this paper, we are going to be a bit more conservative though and only generalize over expressions that are total. This still includes many expressions though, for example, all standard function definitions.

Generalizable bindings are simply written as $x = e$ where the name $x$ is bound to the expression $e$ where the effect of $e$ must be total. The type of a total binding is always generalized and can thus be polymorphic. In contrast, we can bind a name $x$ to the result of an effectful expression as $x \leftarrow e$. The type of such binding is not generalized and has monomorphic type. This corresponds directly to the notion of let bindings and monadic bindings in Haskell.

The reason for only allowing total expressions for let bindings, is that the equal sign in the binding now truly means equality: i.e. such bindings are referentially transparent and we can apply equational reasoning where we can either replace a name by its definition, or abstract an expression into a shared let binding. This clearly cannot be done for effectful functions in general, and not even for effects like divergent, partial, or pure. Consider:

```
x ← 1/0
if true then 1 else x
```

In the above program, we cannot replace the occurence of $x$ by its (partial) binding as that changes the termination behavior of the program: as it stands, it always raises an exception, while the inlined program always returns 1.

Even effectful bindings have many useful properties though. For example, for any expression $e$ with an idempotent effect, the evaluation of the expression (at that program point) will always give the same result, i.e. the expression $\{\, x \leftarrow e; y = x \,\}$ is equivalent to $\{\, x \leftarrow e; y \leftarrow e \,\}$.

### 3.5 On exceptions

Exceptions can occur in any subeffect of the partial effect. Exceptions are raised by undefined matches, intrinsic partial operations like division by zero, or explictly by the user. We can also catch exceptions but we need to be careful. For example, we generally cannot observe exceptions or otherwise our partial effect would no longer be commutative. One way around this issue is to give *catch* an io effect:

$$catchIO : \forall\alpha.\ (() \xrightarrow{\mathsf{io}} \alpha, exn \xrightarrow{\mathsf{io}} \alpha) \xrightarrow{\mathsf{io}} \alpha$$

This works as io already includes non deterministic behaviour, and this is the way Haskell exceptions are catched. Another solution is possible if we make the exact exception unobservable in the handler:

$$catch : \forall\alpha\epsilon.\ (() \xrightarrow{\mathsf{partial}} \alpha, () \xrightarrow{\epsilon} \alpha) \xrightarrow{\epsilon} \alpha$$

We can now for example use a total handler to convert a partial expression into a total one. Moreover, it is possible to use a handler with a st-total($h$) effect for example. Similarly, we can transform pure operations to divergent ones:

$$catchPure : \forall\alpha\epsilon.\ (() \xrightarrow{\mathsf{pure}} \alpha, () \xrightarrow{\mathsf{divergent}} \alpha) \xrightarrow{\mathsf{divergent}} \alpha$$

These two transformations are denoted by the dotted lines in the original effect hierarchy in Figure 1. Of course, we can define similar catch functions for the stateful operations st-partial and st-pure.

In Figure 2 we have not denoted these catch transitions, but we could show them as dotted lines that go horizontally from right to left from each corner in the cubes.

$$
\begin{array}{llll}
\text{Types,Effects} & \tau, \epsilon ::= & \alpha & \text{(type variable)} \\
 & & \mid c(\tau_1, \ldots, \tau_n) & \text{(type constructor )} \\
 & & \mid (\tau_1, \ldots, \tau_n) \xrightarrow{\epsilon} \tau & \text{(function } (n \geqslant 0)) \\
 & & & \\
\text{Type schemes} & \sigma \quad ::= & \forall \overline{\alpha}.\, \tau & \text{(quantification)}
\end{array}
$$

**Figure 3.** Types.

### 3.6 Partiality and non-termination

The reader might be somewhat curious how a type system can determine the partiality and termination effects of an expression. Of course, both the partiality and termination behavior of functions cannot be determined statically in general. Therefore, the type system relies on an separate analysis that determines whether an expression is partial or possibly non-terminating. This analysis is defined according to some simple rules such that the outcome is predictable for the programmer.

Partiality is introduced by the use of partial functions, and by incomplete match expressions. We consider a match only total when it matches on all possible constructors. When guards are involved we require that there is at least one case per constructor that has no guard expression. We are similarly conservative when determining non-termination: a recursive function is only considered to terminate when it uses structural recursion, i.e. in each recursive call, no argument can grow and at least one must be smaller. In contrast to lazy languages like Haskell, this is valid since in a strict setting all data types are inductive and finite.

Clearly, the above decision procedures for partiality and non-termination are quite conservative, but that is fine: for many functions the given rules work well, and we prefer clear conservative rules over complex and unpredictable analysis. One area we would like to handle better is simple recursion over integers where it is useful to detect when integers get smaller. For now, we leave this to future work when we have more experience with larger programs.

## 4. The type system

In this section we are going to treat the type system more formally and explain how types can be derived. As a first step we are going to show a basic type system where all function parameters are annotated with their type. This system is powerful enough to check all examples in this paper and is how we envision how programmers view effect typing. Nevertheless, we would like to have a more powerful inference system where types can be inferred for function parameters too. In Section 4.5 we extend the basic type system with qualified types to allow this.

### 4.1 Types

The grammar of the types is given in Figure 6. Simple *monomorphic types* are written as $\tau$ and consist of type variables, type constructors, and functions. We usually write $\epsilon$ for effect types that form a subset of the monotypes (of kind *Effect*). Type constructors include builtin types like $int$, but also user defined types like $list(\alpha)$. *Type schemes* $\sigma$ have the form $\forall \alpha_1, \ldots, \alpha_n.\, \tau$ where a type $\tau$ is quantified over the type variables $\alpha_1$ to $\alpha_n$.

For the purposes of this paper, type constructors are always fully applied, but the system can be easily extended to also support partially applied type constructors. We do this in our implementation where we use a kind system to ensure that types are well formed (Jones 1995). If partially applied type constructors are allowed, it is important that type variables can no longer range over function arrows ($\rightarrow$). As we will see later, this is vital to keep the simplication of constraints decidable. This property is easy to fulfill by ensuring that a function arrow is always (syntactically) fully applied.

$$
\begin{array}{ll}
\textsc{Sub-Fun} & \dfrac{\tau_i' \leqslant \tau_i \quad \tau \leqslant \tau' \quad \epsilon \leqslant \epsilon'}{(\tau_1, \ldots, \tau_n) \xrightarrow{\epsilon} \tau \leqslant (\tau_1', \ldots, \tau_n') \xrightarrow{\epsilon'} \tau'} \\[2ex]
\textsc{Sub-Trans} & \dfrac{\tau_1 \leqslant \tau_2 \quad \tau_2 \leqslant \tau_3}{\tau_1 \leqslant \tau_3} \\[2ex]
\textsc{Sub-Refl} & \tau \leqslant \tau
\end{array}
$$

**Figure 4.** Structural subtype rules.

$$
\begin{array}{ll}
\textsc{Var} & \dfrac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \mid \mathsf{total}} \\[2ex]
\textsc{Fun} & \dfrac{\Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau \mid \epsilon}{\Gamma \vdash \lambda(x_1 : \tau_1, \ldots, x_n : \tau_n) \rightarrow e : (\tau_1, \ldots, \tau_n) \xrightarrow{\epsilon} \tau \mid \mathsf{total}} \\[2ex]
\textsc{App} & \dfrac{\begin{array}{c}\Gamma \vdash e : (\tau_1, \ldots, \tau_n) \xrightarrow{\epsilon} \tau \mid \epsilon' \quad \Gamma \vdash e_i : \tau_i \mid \epsilon' \\ \mathsf{comm}\, \epsilon' \quad \epsilon' \leqslant \epsilon\end{array}}{\Gamma \vdash e(e_1, \ldots, e_n) : \tau \mid \epsilon} \\[2ex]
\textsc{Let} & \dfrac{\Gamma \vdash e_1 : \sigma_1 \mid \mathsf{total} \quad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash x = e_1; e_2 : \tau_2 \mid \epsilon} \\[2ex]
\textsc{Bind} & \dfrac{\Gamma \vdash e_1 : \tau_1 \mid \epsilon \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash x \leftarrow e_1; e_2 : \tau_2 \mid \epsilon} \\[2ex]
\textsc{Gen} & \dfrac{\Gamma \vdash e : \tau \mid \mathsf{total} \quad \overline{\alpha} \not\cap ftv(\Gamma)}{\Gamma \vdash e : \forall \overline{\alpha}.\, \tau \mid \mathsf{total}} \\[2ex]
\textsc{Inst} & \dfrac{\Gamma \vdash e : \forall \overline{\alpha}.\, \tau \mid \epsilon}{\Gamma \vdash e : [\overline{\alpha} := \overline{\tau}]\tau \mid \epsilon} \\[2ex]
\textsc{Sub} & \dfrac{\Gamma \vdash e : \tau_1 \mid \epsilon \quad \tau_1 \leqslant \tau_2}{\Gamma \vdash e : \tau_2 \mid \epsilon} \\[2ex]
\textsc{Lift} & \dfrac{\Gamma \vdash e : \sigma \mid \epsilon_1 \quad \epsilon_1 \leqslant \epsilon_2}{\Gamma \vdash e : \sigma \mid \epsilon_2} \\[2ex]
\textsc{Run-Pure} & \dfrac{\Gamma \vdash e : \sigma \mid \mathsf{st\text{-}pure}(h) \quad h \notin ftv(\sigma, \Gamma)}{\Gamma \vdash \mathsf{run}\ e : \sigma \mid \mathsf{pure}}
\end{array}
$$

**Figure 5.** Type rules.

Furthermore, we have two constraint relations on types and effects: the predicate $\tau_1 \leqslant \tau_2$ constrains a type $\tau_2$ to be a subtype of $\tau_1$, and the predicate $\mathsf{comm}\, \epsilon$ constrains $\epsilon$ to a commutative effect. We assume that all commutative effects are defined as axioms. In our case, it holds for the effects total, partial, divergent, read-total, read-partial, read-divergent, alloc-total, alloc-partial, and alloc-divergent.

The subtype relation $\tau_1 \leqslant \tau_2$ is assumed to form a semilattice with total as the least element. Moreover, the subtype rules in Figure 4 should hold. We assume that all standard subtype relations are added as defined in Figure 1 and Figure 2, and includes for example $\mathsf{partial} \leqslant \mathsf{pure}$, $\mathsf{read\text{-}total}(h) \leqslant \mathsf{st\text{-}total}(h)$, and $\mathsf{read\text{-}total}(h) \leqslant \mathsf{read\text{-}partial}(h)$.

In Figure 4, the top rule, Sub-Fun, encodes the usual subtyping relation on functions where the subtype relation switches for arguments. Because functions can be nested, we usually say that a type is in a positive position when it can be increased to a subtype, and in a negative position if the direction is opposite. If a type is in both a negative and positive position, we call it neutral or invariant. The next two rules define that subtyping is transitive and reflexive.

### 4.2 Type rules

Before we discuss the type rules, we first define the core language of expressions. The grammar is defined as:

$$\begin{array}{rll} e ::= & x & \text{(variable )} \\ \mid & e(e_1, ..., e_n) & \text{(function application)} \\ \mid & \lambda(x_1, ..., x_n) \rightarrow e & \text{(function definition)} \\ \mid & x = e_1; e_2 & \text{(let binding)} \\ \mid & x \leftarrow e_1; e_2 & \text{(effect binding)} \\ \mid & \mathsf{run}\ e & \text{(run isolated state)} \end{array}$$

Figure 5 gives the type rules of our system. These rules are declarative and are not syntax directed. The declaration $\Gamma \vdash e : \tau \mid \epsilon$ states that expression $e$ can be assigned the type $\tau$ with effect $\epsilon$, assuming a type environment $\Gamma$. The type environment maps free variables to types. We write environment extension as $(\Gamma, x : \sigma)$ which removes any previous bindings for $x$ from $\Gamma$ and adds the binding $x : \sigma$ to $\Gamma$. The inferred effect $\epsilon$ means that evaluation of the expression $e$ can have a potential effect $\epsilon$.

The first rule VAR simply states that the type for a variable $x$ is the type bound in the environment. Since there is no evaluation for a variable, we can assign a total effect. The next rule FUN is for lambda expressions. Since evaluation of the *definition* of a function has no effect at all, the inferred effect is total: indeed, functions are just values. The inferred effect of the body of the function in the premise shifts to the arrow of the function type in the conclusion, since *calling* the function will lead to evaluation of the body. Note that we assume that all function parameters are annotated with their type. This ensures that we have principle types and greatly simplifies type inference to be almost like type checking since only result types are inferred.

The rule for applications is the most interesting with regard to effects. In particular, the effects of the arguments $e_i$ and the function expression $e$ are restricted to commutative effects. Since commutative effects form a semi-lattice we ensure this by requiring that $e_i$ and $e$ have a common effect $\epsilon'$ which is commutative, i.e. $\mathsf{comm}\,\epsilon'$. Note that we can always lift all arguments to a common effect through the rule LIFT. The effect assigned to the function application is of course the effect of the function: dual to function definitions, the effect on the arrow in the premise now shifts to the effect inferred in the conclusion. Also, we need to ensure that this effect is a subtype of the effect of the arguments and function expression $\epsilon'$, i.e. $\epsilon' \leqslant \epsilon$. Without this premise, we might for example have a divergent argument but infer a total effect which would be wrong. In the case that $\epsilon \leqslant \epsilon'$, we can always lift the type of a function using rule SUB to satisfy this premise.

The next two rules deal with binding names to expressions. The LET rule allows names to be bound to a polymorphic type $\sigma$, but requires that the effect of the body of the definition is total. In contrast, the rule BIND binds names to expressions with an arbitrary effect but restricts the binding to monomorphic types $\tau$.

Rule GEN is the usual generalization rule where a type can be generalized over variables $\overline{\alpha}$ as long as $\overline{\alpha}$ is disjoint with the free type variables of the environment. Note that we can only generalize over expressions that are total. As shown in Section 3.4 it would be unsafe to allow generalization over arbitrary side effecting operations.

Rule INST instantiates quantifiers to a monomorphic type. The next rule SUB states that if we can derive a type $\tau$, we can also derive a type $\tau'$ if that is a subtype of $\tau$, i.e. $\tau \leqslant \tau'$.

The last two rules are specific to effects. The rule LIFT allows us to use subtyping on the derived effect: if we can derive an effect $\epsilon_1$, we can also derive a 'worse' effect $\epsilon_2$ if these are in a subtype relation: $\epsilon_1 \leqslant \epsilon_2$. The rule RUN-PURE captures those stateful expressions that can be considered pure since their heap is unobservable from outside. If the heap variable $h$ is not among the free type variables in the assumption, or type, we can safely derive the effect pure. For simplicity we left out the other three variants of this rule for total, partial, and divergent heap effects.

### 4.3 Divergence

We assume that primitive effect operations like throwning an exception, reference creation and assigment etc. are defined as primitive functions, similarly to stateful operations in Haskell.

One effect that cannot be described that way is divergence. Instead we assume that the termination analysis phase translates recursive let-bindings to a special syntactic construct, $\mathsf{rec}\ f : \overline{\tau} \xrightarrow{\epsilon} \tau = e$ where we assume for simplicity that such recursive bindings are annotated with a monomorphic type (but in practice it is straightforward to infer polymorphic types for monomorphic recursion). Since we have strict evaluation the type must be a function, and since it is recursive, we require that the effect $\epsilon$ is a divergent effect, like divergent or $\mathsf{st\text{-}divergent}(h)$ for example. The type rule for this binding is now straightforward:

$$\textsc{Let-Rec}\quad \frac{\Gamma, f : \overline{\tau} \xrightarrow{\epsilon} \tau \vdash e_1 : \overline{\tau} \xrightarrow{\epsilon} \tau \mid \mathsf{total} \quad \epsilon \text{ is divergent} \qquad \Gamma, f : \overline{\tau} \xrightarrow{\epsilon} \tau \vdash e_2 : \sigma \mid \epsilon'}{\Gamma \vdash \mathsf{rec}\ f : \overline{\tau} \xrightarrow{\epsilon} \tau = e_1; e_2 : \sigma \mid \epsilon'}$$

### 4.4 Type inference

As remarked, since all function parameters are annotated with their type, type checking is straightforward. On the other hand, it can be a burden to annotate all function parameters and we would like to have a type inference system that needs no type annotations on function parameters.

Unfortunately, once we remove such annotations, we can no longer assign principal types to expressions. The reason for this is that we lack polymorphic subtype and commutativity constraints. Take for example the function *twice* defined as:

$$\mathit{twice}(f : \alpha \xrightarrow{\mathsf{partial}} \alpha, x : \alpha) = f(f(x))$$

Here the user annotated the function $f$ with a partial effect to satisfy the commutivity constraint arising from the application $f(f(x))$, but a divergent effect would suffice too:

$$\mathit{twice}(f : \alpha \xrightarrow{\mathsf{divergent}} \alpha, x : \alpha) = f(f(x))$$

Unfortunately, without annotation we cannot choose a principal (or 'best') type. In order to be able to give principal types, we are going to use polymorphic *qualified types*. In particular, the type of *twice* can then be inferred as:

$$\mathit{twice} : \forall \alpha\epsilon.\ \mathsf{comm}\,\epsilon \Rightarrow (\alpha \xrightarrow{\epsilon} \alpha, \alpha) \xrightarrow{\epsilon} \alpha$$

or as its principal type, namely:

$$\mathit{twice} : \forall \alpha\beta\epsilon.\ (\beta \leqslant \alpha, \mathsf{comm}\,\epsilon) \Rightarrow (\alpha \xrightarrow{\epsilon} \beta, \alpha) \xrightarrow{\epsilon} \beta$$

In the next section, we extend the basic type system with qualified types to enable type inference with principal types. At the same time we still restrict user annotations to types without subtype constraints; a choice we will motivate in Section 4.9.

### 4.5 An extension with qualified types

Figure 6 shows our types extended with qualifiers. *Type schemes* $\sigma$ now have the form $\forall \alpha_1, ..., \alpha_n.\ (\pi_1, ..., \pi_n) \Rightarrow \tau$ where a type $\tau$ is quantified over the type variables $\alpha_1$ to $\alpha_n$. The *qualifiers* $\pi_1$ to $\pi_m$ are predicates that need to hold.

We write a set of predicates as P. There is an *entailment* relation written as $\mathrm{P}_1 \Vdash \mathrm{P}_2$ which asserts that the predicates in $\mathrm{P}_2$ hold whenever those in $\mathrm{P}_1$ are satisfied. Following Jones (1992) We require that this relation is transitive, closed under substitution, and that $\mathrm{P}_1 \Vdash \mathrm{P}_2$ whenever $\mathrm{P}_2 \subseteq \mathrm{P}_1$.

$$\begin{array}{llll}
\text{Qualified types} & \sigma & ::= \forall\overline{\alpha}.\,\overline{\pi} \Rightarrow \tau & \text{(quantify)} \\
 & & & \\
\text{Qualifiers} & \mathrm{P}, \overline{\pi} & ::= \pi_1, ..., \pi_n & \text{(predicates )} \\
 & \pi & ::= \tau_1 \leqslant \tau_2 & \text{(sub type)} \\
 & & \mid \ \mathsf{comm}\,\epsilon & \text{(commutative)}
\end{array}$$

**Figure 6.** Type schemes with qualified types.

ENTAIL
$$\frac{\pi \in \mathrm{P}}{\mathrm{P} \Vdash \pi}$$

SUB-FUN
$$\mathrm{P} \Vdash \tau_i' \leqslant \tau_i \quad \mathrm{P} \Vdash \tau \leqslant \tau' \quad \mathrm{P} \Vdash \epsilon \leqslant \epsilon'$$

ular, according to the subtype relation in Figure 7 data types are always neutral and that only function types have nested subtyping. This means in practice that any constraint $\alpha \leqslant c(\tau_1, ..., \tau_n)$ for example can be simplified to $\alpha = c(\tau_1, ..., \tau_n)$ removing many constraints that would arise in systems with subtyping on general types.

This also implies that we can always bring subtype constraints into an *atomic form* where subtype is always between either a type variable or effect constant, for example, $\alpha \leqslant \beta$, or $\epsilon_1 \leqslant \epsilon_2$. As an aside, in our implementation we allow higher order polymorphism where type constructors can be partially applied. We restrict polymorphism to range only over neutral type constructors and not over the function arrow. As a result, we can still simplify constraints of the form $\alpha \leqslant \beta(\tau_1, ..., \tau_n)$ since it is guaranteed that there will be no nested subtyping (since $\beta$ can never be substituted with the arrow constructor).

Moreover, since we do have a deep subtype relation over functions, it is possible to simplify many types where part of the constraint only occurs in a positive or negative location. Take for example the *apply* function:

$$apply(f, x)\,\{f(x);\,\}$$

where we can derive a principal type:

$$\forall\alpha\beta\alpha_1\beta_1\epsilon\epsilon_1.\,(\epsilon \leqslant \epsilon_1, \alpha_1 \leqslant \alpha, \beta \leqslant \beta_1) \Rightarrow (\alpha \xrightarrow{\epsilon} \beta, \alpha_1) \xrightarrow{\epsilon_1} \beta_1$$

which is quite complicated. However, since $\beta_1$ and $\epsilon_1$ occur only in a positive position and are only bound below, we can simplify them to equal their lower bounds, $\beta = \beta_1$ and $\epsilon = \epsilon_1$. Likewise, since $\alpha_1$ occurs only negatively and is only bound above, we can make it equal to its upper bound $\alpha = \alpha_1$. The simplified type will be:

$$\forall\alpha\beta.\,(\alpha \xrightarrow{\epsilon} \beta, \alpha) \xrightarrow{\epsilon} \beta$$

which is the same as the standard Hindley-Milner type. Note that under our instance relation (Section 4.6) these two types are *equivalent* and one type is not better than the other in a semantic sense.

In general there might be constraints that cannot be simplified any further. For example, the principal type for *twice*, defined as:

$$twice(f, x)\,\{f(f(x));\,\}$$

is

$$\forall\alpha\beta\epsilon.\,(\beta \leqslant \alpha, \mathsf{comm}\,\epsilon) \Rightarrow (\alpha \xrightarrow{\epsilon} \beta, \alpha) \xrightarrow{\epsilon} \beta$$

For this particular type, both $\alpha$ and $\beta$ are neutral and we cannot simplify it further.

### 4.9 User annotations and decidability

The system as described does not allow type annotations on let bindings, but in practice this is of course required. Unfortunately, we cannot support general user annotations but need to restrict the type schemes that the user can write to type schemes without any subtype constraints (but commutativity constraints are allowed).

The reason for this restriction is that it is generally not decidable whether the subtype constraints in a user defined type annotation imply the inferred subtype constraints (Pottier 1996; Smith 1991). Even in the restricted case where subtype constraints consist just of type variables and where the effect hierarchy is finite, this is a NP hard problem.

Fortunately, the simplification procedure sketched in the previous section illustrates that the type of most functions can be reduced to contain no subtype constraints at all. In our experience, the remaining functions such as *twice* are often intended to have a polymorphic type without contraints as well. In light of this, we chose to disallow any subtype constraints in user defined type annotations. We still allow commutativity constraints though since constraint implication for these is decidable. For example, we can *not* annotate *twice* with its principal type, but we can annotate it with a slightly more restrictive type that has no subtype constraint:

$$twice : \forall\alpha\epsilon.\,(\mathsf{comm}\,\epsilon) \Rightarrow (\alpha \xrightarrow{\epsilon} \alpha, \alpha) \xrightarrow{\epsilon} \alpha$$

This type is probably expected in practice anyway since the one with the subtype constraint is generally difficult to envision for most people. With this restriction in place predicate solving becomes efficient and can purely be done through local reasoning, and as a consequence type inference is decidable with these restricted annotations.

Obviously, the weakness of our current approach is that even though the system infers principal types, the user might not be able to actually write that type down. Moreover, for some programs, annotating with a more restrictive type would make the program no longer type check.

We would love to improve on this but we have not found a satisfactory solution. On the other hand, we have not seen any programs yet where this is a problem in practice: we argue that this is a consequence of two reasons. First, the subtype relation is very limited in scope and only applies to effects (and not to general data types). Secondly, the use of effects generally leads to the effects in positive positions. The exception are effectful higher-order functions, but those are generally either applied positively, or passed directly to other functions. The cases where a higher order argument is applied to itself (as in *twice*), or used in multiple negative positions, occur seldomly.

## 5. A type-directed monadic translation

Figure 9 defines a type-directed translation from the basic type system to a predicative fragment of System F. Our translation assumes the existence of a monad $M_\epsilon$ for each effect $\epsilon$, including the primitive monad operations that should satisfy the monad laws:

$$\begin{array}{lll} \gg\!\!=_\epsilon & : \forall\alpha\beta.\, M_\epsilon(\alpha) \to (\alpha \to M_\epsilon(\beta)) \to M_\epsilon(\beta) & \text{(bind)} \\ unit_\epsilon & : \forall\alpha.\, \alpha \to M_\epsilon(\alpha) & \text{(return)} \end{array}$$

Our translation also assumes a family of run functions for each basic effect. For example:

$$run_{\mathsf{pure}} : \forall\alpha.\,(\forall h.\, M_{\mathsf{st\text{-}pure}(h)}(\alpha)) \to M_{\mathsf{pure}}(\alpha) \quad \text{(runST)}$$

We treat the total monad specially though and treat $M_{\mathsf{total}}(\sigma)$ as a type synonym for $\sigma$, i.e. total is the identity monad. As such, the bind operation for total is just reverse function application and the unit operation is the identity function. For simplicity, we do not translate types explicitly to System-F types but directly interpret a type like $\tau_1 \xrightarrow{\epsilon} \tau_2$ as $\tau_1 \to M_\epsilon(\tau_2)$ when necessary.

For each basic subeffect relation, we assume that a proper evidence function f exists, written as $\epsilon_1 \leqslant \epsilon_2 \rightsquigarrow \mathsf{f}$ where the evidence function f has type $\forall\alpha.\, M_{\epsilon_1}(\alpha) \to M_{\epsilon_2}(\alpha)$. Using the structural subtype rules from Figure 4 it is straightforward to define the subtype relation on types $\tau_1 \leqslant \tau_2 \rightsquigarrow \mathsf{f}$ where the evidence function f has type $\tau_1 \to \tau_2$. For example, the transitivity rule simply composes evidence:

$$\textsc{Sub-Trans}\quad \frac{\tau_1 \leqslant \tau_2 \rightsquigarrow \mathsf{f}_1 \quad \tau_2 \leqslant \tau_3 \rightsquigarrow \mathsf{f}_2}{\tau_1 \leqslant \tau_3 \rightsquigarrow \mathsf{f}_2 \circ \mathsf{f}_1}$$

Using these primitives, the monadic type directed translation is straigtforward and follows exactly the structure of the type derivation. Each rule $\Gamma \vdash e : \sigma \mid \epsilon \rightsquigarrow \mathsf{e}$ states that if we derive a type $\sigma$ with effect $\epsilon$ for an expression $e$ under the environment $\Gamma$, we can also derive a corresponding well-typed System-F expression e with type $M_\epsilon(\sigma)$. Well-typedness of the System F expression is straightforward to prove by induction over the type rules. Note how expressions with a total effect, like in the rules Var, Fun, and Gen translate directly to their corresponding System-F expressions.

$$\textsc{Var}\quad \frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \mid \mathsf{total} \rightsquigarrow \mathsf{x}}$$

$$\textsc{Fun}\ \frac{\Gamma, x_1 : \tau_1, ..., x_n : \tau_n \vdash e : \tau \mid \epsilon \rightsquigarrow \mathsf{e}}{\begin{array}{c}\Gamma \vdash \lambda(x_1, ..., x_n) \to e : (x : \tau_1, ..., x : \tau_n) \xrightarrow{\epsilon} \tau \mid \mathsf{total} \\ \rightsquigarrow \lambda(x_1 : \tau_1, ..., x_n : \tau_n).\, \mathsf{e}\end{array}}$$

$$\textsc{App}\ \frac{\begin{array}{c}\Gamma \vdash e : (\tau_1, ..., \tau_n) \xrightarrow{\epsilon} \tau \mid \epsilon' \rightsquigarrow \mathsf{e} \quad \Gamma \vdash e_i : \tau_i \mid \epsilon' \rightsquigarrow \mathsf{e}_i \\ \mathsf{comm}\,\epsilon' \quad \epsilon' \leqslant \epsilon \rightsquigarrow \mathsf{f}\end{array}}{\begin{array}{l}\Gamma \vdash e(e_1, ..., e_n) : \tau \mid \epsilon \rightsquigarrow \\ \quad \mathsf{f}(\mathsf{e}) \quad \gg\!=_\epsilon \ \lambda(\mathsf{x} : (\tau_1, ..., \tau_n) \xrightarrow{\epsilon} \tau). \\ \quad \mathsf{f}(\mathsf{e}_1) \ \gg\!=_\epsilon \ \lambda(\mathsf{x}_1 : \tau_1). \\ \qquad\qquad ... \\ \quad \mathsf{f}(\mathsf{e}_n) \ \gg\!=_\epsilon \ \lambda(\mathsf{x}_n : \tau_n). \\ \quad \mathsf{x}(\mathsf{x}_1, ..., \mathsf{x}_n)\end{array}}$$

$$\textsc{Let}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma_1 \mid \mathsf{total} \rightsquigarrow \mathsf{e}_1 \\ \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathsf{e}_2\end{array}}{\Gamma \vdash x = e_1; e_2 : \tau_2 \mid \epsilon \rightsquigarrow (\lambda(\mathsf{x} : \sigma_1).\, \mathsf{e}_2)\ \mathsf{e}_1}$$

$$\textsc{Bind}\quad \frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \mid \epsilon \rightsquigarrow \mathsf{e}_1 \\ \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathsf{e}_2\end{array}}{\Gamma \vdash x \leftarrow e_1; e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathsf{e}_1 \gg\!=_\epsilon \lambda(\mathsf{x} : \tau_1).\, \mathsf{e}_2}$$

$$\textsc{Gen}\quad \frac{\Gamma \vdash e : \tau \mid \mathsf{total} \rightsquigarrow \mathsf{e} \quad \overline{\alpha} \not\cap \mathit{ftv}(\Gamma)}{\Gamma \vdash e : \forall \overline{\alpha}.\, \tau \mid \mathsf{total} \rightsquigarrow \Lambda(\alpha_1, ..., \alpha_n).\, \mathsf{e}}$$

$$\textsc{Inst}\quad \frac{\Gamma \vdash e : \forall \overline{\alpha}.\, \tau \mid \epsilon \rightsquigarrow \mathsf{e} \quad \overline{\tau} = \tau_1, ..., \tau_n}{\Gamma \vdash e : [\overline{\alpha} := \overline{\tau}]\tau \mid \epsilon \rightsquigarrow \mathsf{e}[\tau_1, ..., \tau_n]}$$

$$\textsc{Sub}\quad \frac{\Gamma \vdash e : \tau_1 \mid \epsilon \rightsquigarrow \mathsf{e} \quad \tau_1 \leqslant \tau_2 \rightsquigarrow \mathsf{f}}{\Gamma \vdash e : \tau_2 \mid \epsilon \rightsquigarrow \mathsf{e} \gg\!=_\epsilon (\mathit{unit}_\epsilon \circ \mathsf{f})}$$

$$\textsc{Lift}\quad \frac{\Gamma \vdash e : \sigma \mid \epsilon_1 \rightsquigarrow \mathsf{e} \quad \epsilon_1 \leqslant \epsilon_2 \rightsquigarrow \mathsf{f}}{\Gamma \vdash e : \sigma \mid \epsilon_2 \rightsquigarrow \mathsf{f}(\mathsf{e})}$$

$$\textsc{Run-Pure}\quad \frac{\Gamma \vdash e : \sigma \mid \mathsf{st\text{-}pure}(h) \rightsquigarrow \mathsf{e} \quad h \notin \mathit{ftv}(\sigma, \Gamma)}{\Gamma \vdash \mathsf{run}\ e : \sigma \mid \mathsf{pure} \rightsquigarrow \mathit{run}_{\mathsf{pure}}\ (\Lambda h.\, \mathsf{e})}$$

**Figure 9.** Monadic type-directed translation.

Since System-F is sound, we have as a consequence that our basic type system is sound, under the assumption that our primitive monad operations are sound. We define these remaining operations within the next section.

## 6. Unambiguous denotational semantics

Our operational semantic definition in the previous section is rather weak: the monads are still abstract and there are more possible translations that are also correct, for example with respect to the order of argument evaluation. In this section we will make the semantics more precise.

Generally, operational semantics essentially specify an implementation of the language, but they do not assign any deeper meaning to programs in that language. For example, programmers and compilers might reason that the program $x * 0$ is equivalent to 0. However, in Haskell for example, "*" does not mean multiplication exactly, and variable $x$ of type $\mathit{int}$ does not quite mean a 32-bit integer, so these programs are not actually equivalent in Haskell.

We want to give types and programs a precise natural meaning. In particular, a program of type $\mathit{int} \xrightarrow{\mathsf{total}} \mathit{int}$ means precisely a total function on 32-bit integers. To formalize this concept, we use constructive set theory to specify a denotational semantics for our language. The System-F translation above does part of the work, since System-F without recursion can be modeled directly by constructive set theory. The only remaining components are the definitions of the monad and monad operations for each effect, which are presented later in this section. The reason why we don't use System-F for all definitions is that certain effects, like divergence and I/O, cannot be expressed directly in System-F's type system as such, but we can define them in constructive set theory.

### 6.1 Effects

Our starting point for the denotational semantics is the type directed translation to System-F in Figure 9. With the difference that we interpret the System-F program in constructive set theory, where all functions are total by definition. Moreover, we will show that for each of the monads we will define, the corresponding monad laws hold. In particular:

$$\begin{array}{lll}(\textsc{M-Left}) & \mathit{unit}_\epsilon(x) \gg\!=_\epsilon f & \equiv \ f(x) \\ (\textsc{M-Right}) & e \gg\!=_\epsilon \mathit{unit}_\epsilon & \equiv \ e \\ (\textsc{M-Assoc}) & e \gg\!=_\epsilon (\lambda x. f(x) \gg\!=_\epsilon g) & \equiv \ (e \gg\!=_\epsilon f) \gg\!=_\epsilon g\end{array}$$

We rely on the monad-assocativity law M-Assoc to resolve ambiguity in how to parse a sequence of commands. In particular, the following program has the following two possible translations:

$$\begin{array}{l}e_1; \\ e_2; \\ e_3\end{array} \ \xmapsto{\text{translates to}} \ \begin{array}{c}e_1 \gg\!= (\lambda\_.e_2 \gg\!= (\lambda\_.e_3)) \\ \text{or} \\ (e_1 \gg\!= (\lambda\_.e_2)) \gg\!= (\lambda\_.e_3)\end{array}$$

Essentially the program could be parsed as $e_1; (e_2; e_3)$ or $(e_1; e_2); e_3$ where the monad-associativity law guarantees that the two translations are semantically equivalent.

### 6.2 Subeffects

A major component of our language is its ability to mix different effects together by using the concept of subeffects. Languages with subtypes often have an underlying function which shows how to convert the subtype into the supertype. Often this process is injective or monomorphic, meaning that distinct values in the subtype remain distinct in the supertype. We take a similar approach where for every two subeffects $\epsilon_1 \leqslant \epsilon_2$ we have an injective natural transformation $\mathsf{f}$ with type $\forall\alpha.\, M_{\epsilon_1}(\alpha) \to M_{\epsilon_2}(\alpha)$. For clarity, we will often write the function $\mathsf{f}$ as $\mathit{sub}_{\epsilon_1 \leqslant \epsilon_2}$.

We have to be careful with our choice of natural transformations, though, or else our language can have an ambiguous semantics. This can be quite challenging, and so we take inspiration from Reynolds' concept of natural subtyping (Reynolds 1980). Suppose we have the expression 1+2+"hi", using "+" as both addition and as string concatenation. If we assume that integers are a subtype of strings, we haven an ambigious expression that can evaluate both to "12hi" or "3hi".

We avoid this situation by making subeffecting natural by construction. In particular, we ensure that all the primitive lifting functions $\mathsf{f}$ with type $\forall\alpha.\, M_{\epsilon_1}(\alpha) \to M_{\epsilon_2}(\alpha)$ satisfy the monad morphism laws (Liang et al. 1995; Wadler 1992):

$$\begin{array}{ll}(\textsc{M-Unit}) & \mathsf{f} \circ \mathit{unit}_{\epsilon_1} \quad\ \ \equiv \mathit{unit}_{\epsilon_2} \\ (\textsc{M-Bind}) & \mathsf{f}\ (e \gg\!=_{\epsilon_1} g) \equiv \mathsf{f}(e) \gg\!=_{\epsilon_2} (\mathsf{f} \circ g)\end{array}$$

There are many ways in which one effect may be a subeffect of another. If we consider the structural subtype rules, we see that whenever we have $\epsilon \leqslant \epsilon'$ and $\epsilon' \leqslant \epsilon''$, we can derive that $\epsilon$ is a subeffect of $\epsilon''$ through transitivity. This leaves us with two ways to convert values with effect $\epsilon$ to have effect $\epsilon''$: use the indirect path of applying $\mathit{sub}_{\epsilon \leqslant \epsilon'}$ followed by $\mathit{sub}_{\epsilon' \leqslant \epsilon''}$, or use the direct path of simply applying $\mathit{sub}_{\epsilon \leqslant \epsilon''}$. For our language to be unambiguous, we ensure that both methods always produce the same result. This is guaranteed by ensuring M-Unit holds for these definitions. Similarly, the reflexivity law makes any effect a subeffect of itself, so we always define $sub_{\epsilon \leqslant \epsilon}$ as the identity function, again ensuring an unambiguous semantics.

More significantly, we must ensure that subeffects interact unambiguously with how we sequence effectful computations. Suppose we have a sequence of two effectful computations both with effect $\epsilon$ which is a subeffect of $\epsilon'$. There are two ways in which we might convert this sequence so that it has effect $\epsilon'$:

$$\begin{array}{l} x \leftarrow sub(e_1); \\ sub(e_2) \end{array} \quad \text{or } sub \left( \begin{array}{l} x \leftarrow e_1; \\ e_2 \end{array} \right)$$

That is, we may either convert their effect before or after we sequence them. Again, by enforcing the monad morphism laws we ensure that either process produces the same result.

There is an interesting consequence of using subeffects: we have no need for the `return` function used in Haskell to explicitly convert noneffectful computations into effectful computations. Instead, this is accomplished implicitly by the fact that every effect is a supereffect of total. In fact, for any effect $\epsilon$, the function $sub_{\mathsf{total} \leqslant \epsilon}$ converting from total to $\epsilon$ is the unit of the monad $M_\epsilon$. The monad-identity laws ensure that this is a monad morphism.

### 6.3 The Monads

So far we have set up the framework for an effectful language with unambiguous denotational semantics. At this point, there is still a key component missing: the actual monads and monad morphism definitions. Here we outline the monad used for each effect, and discuss some of the monad morphisms used for conversion. We also phrase our primitive operations in terms of these monads, demonstrating their soundness. Thus, soundness of our language becomes immediate consequence of our denotational semantics.

**total** The total effect is represented using the identity monad. In fact, we can take a type $M_{\mathsf{total}}(\tau)$ as a synonym for $\tau$, and give trivial implementations for the unit and bind operations:

$$unit_{\mathsf{total}}(x) = x$$

$$e \gg\!\!=_{\mathsf{total}} f = f(e)$$

Thus any function with this effect corresponds directly to a total mathematical function. The proof that this monad is commutative is trivial.

**partial** The partial effect is defined as usual:

$$M_{\mathsf{partial}}(\tau) = Ok(\tau) + Fail$$

$$unit_{\mathsf{partial}}(x) \qquad = Ok(x)$$

$$\begin{array}{l} Ok(e) \gg\!\!=_{\mathsf{partial}} f = \mathrm{f}(e) \\ Fail \gg\!\!=_{\mathsf{partial}} f \quad = Fail \end{array}$$

This is a commutative monad because there is only one way to fail. Using this monad we can define partial operations. For example, integer division can be defined as

$$i \div j = \begin{cases} j = 0 & Fail \\ \text{otherwise} & Ok(\lfloor {}^{i}/_{j} \rfloor) \end{cases}$$

Also, whenever cases are missing from a pattern match, we can simply implicitly map the missing cases to $Fail$. Thus, we ensure that all our introductions of the partial effect are sound.

We can also use this monad to show how we can eliminate the partial effect. Specifically, we can define the $catch$ operation as:

$$\begin{array}{l} catch : (thrower : () \xrightarrow{\mathsf{partial}} \tau, handler : () \xrightarrow{\epsilon} \tau) \xrightarrow{\epsilon} \tau \\ \quad = \text{case } thrower() \text{ of} \begin{cases} Ok(x) & unit_\epsilon(x) \\ Fail & handler() \end{cases} \end{array}$$

which shows our elimination of the partial effect is also sound.

**divergent** The divergent effect is represented using the non-termination monad $M_{\mathsf{divergent}}$ We can represent this monad in various ways in constructive set theory, using for example co-algebras. The details are beyond the scope of this paper, and are not so important here since we never eliminate the divergent effect; all that matters is that it can be used to encode arbitrary recursive functions. In constructive set theories, $M_{\mathsf{divergent}}$ is different from $M_{\mathsf{partial}}$; In particular we cannot recognize a value as having the divergence condition in contrast to partiality; thus, there is no $catch$ counterpart for divergence. This monad is commutative, since if either effectful expression fails to terminate, then neither sequencing of the expressions will terminate. As is usual, we often write the application of a divergent monad $M_{\mathsf{divergent}}(\tau)$ as $\tau_\perp$.

**pure** The pure effect combines both the partial and divergent effects into a single monad:

$$M_{\mathsf{pure}}(\tau) = (M_{\mathsf{partial}}(\tau))_\perp$$

We can define it by using the following distributive law (King and Wadler 1993), which defines a nice way for two monads to interact in order to give their composition a monadic structure:

$$dist : \forall \alpha.\, M_{\mathsf{partial}}(\alpha_\perp) \to M_{\mathsf{pure}}(\alpha)$$

$$\begin{array}{ll} dist(Ok(dx)) & = dx \;\gg\!\!=_\perp\; (unit_\perp \circ Ok) \\ dist(Fail) & = unit_\perp(Fail) \end{array}$$

We can also define $catchPure$ by lifting $catch$ to propogate any non-termination, turning any pure computation into a divergent one.

Unlike our earlier monads, the pure monad is not commutative. The reason is that there are now two ways to fail: throw an exception or never terminate. If we change the order of effectful computations, we might change which of these two failures occurs. In many languages, like Haskell for example, the divergent and partial effect are merged into a single failure condition (which is non-deterministic) to simplify the semantics but it is less precise.

**heap** Modeling heap effects faithfully in constructive set theory proved to be a challenge and required some complex type-theoretic constructs: in particular, we need a special heap type where the type itself can be updated to faithfully model allocation. Explaining these new constructs goes beyond the scope of the paper, so for explanation purposes we treat heaps as finite maps from integer locations to values. This model is slightly inaccurate though as it assumes that only valid and appropriately typed references are read. For our language this property holds though as shown by Launchbury and Sabry (1997).

Let $H$ be the type for heaps: finite maps from integer locations to values. For sake of brevity, we only present the type component for the three basic effects read-total, alloc-total, and write-total but we will present st-pure in full detail though, as it is an especially interesting case. The types of the three basic effects are:

$$M_{\mathsf{read\text{-}total}(h)}(\tau) = H \to \tau$$

$$M_{\mathsf{write\text{-}total}(h)}(\tau) = H \times \tau$$

$$M_{\mathsf{alloc\text{-}total}(h)}(\tau) = \mathbb{N} \to H \times \tau$$

Since read-total only reads the heap, the monad is just a function from the heap $H$ to its result. Dually, write-total only writes and takes no $H$ parameters. Finally alloc-total takes a natural number as an argument to be able to assign a fresh location in the returned heap $H$. The order in which operations read from the heap have no impact on their results, so reading is commutative provided that the underlying effect is also commutative, i.e. read-total, read-partial, and read-divergent.

Allocation is not commutative perse due to the integer argument used to allocate fresh locations. However, since these locations are opaque, the numbers are not observable outside this effect, and we can consider allocation commutative modulo the values of these otherwise unobservable numbers (as long as the underlying effect is also commutative).

One may expect write-total to be defined using $H \rightarrow H$, but this does not reflect the fact that the change to the heap is independent of contents of the heap. Instead, we encode write with a partial heap indicating the locations and values that were written. The bind operation combines the two partial heaps by preferring values in the second heap (representing the more recent writes). The write effect is not commutative, since it is important that the most recent writes to the heap are preferred.

Finally, we define the monad for st-pure, which combines all the heap effects along with the pure effect.

$$M_{\mathsf{st\text{-}pure}(h)}(\tau) = H \rightarrow (H \times M_{\mathsf{partial}}(\tau))_\bot$$

$$unit_{(\mathsf{st\text{-}pure}(h))}(x) = \lambda heap.\ unit_\bot(\langle heap, unit_{\mathsf{partial}}(x)\rangle)$$

$$\begin{aligned} e \gg\!=_{(\mathsf{st\text{-}pure}(h))} f \\ = \lambda heap.\ (e\ heap)\ \gg\!=_\bot \lambda\langle heap', px\rangle. \\ \textbf{case}\ px\ \textbf{of} \\ Ok(x) \rightarrow f(x)(heap') \\ Fail \quad \rightarrow unit_\bot(\langle heap', Fail\rangle) \end{aligned}$$

The bind operation passes the heap resulting from the first effectful value into the second effectful operation. As a consequence this effect is not commutative. Moreover, if the first effectful value failed however, then we simply use the resulting heap and completely bypass the second operation entirely. Thus, even in failure we propagate all changes to the heap, as one usually expects. Also note that the partial effect is applied to $\tau$ whereas the divergent effect is applied to the whole tuple. Thus, the two components of the pure effect are split up when combined with the heap effects which is necessary to produce the expected semantics of propagating changes to the heap even in light of failure.

Originally, we had designed the heap effects as monad transformers $\mathsf{st}(h, \epsilon)$ parameterised by a nested effect $\epsilon$, for example $\mathsf{st}(h, \mathsf{total})$. We also allowed operations that were polymorphic with respect to this nested effect. However, when we tried to define the semantics as described in this section, we discovered that $\mathsf{st}(h, \mathsf{pure})$ could not be given the intended semantics (and likewise for partial). The best approximation we can get is the following type for an st effect that is polymorphic in its nested effect:

$$M_{\mathsf{st}(h,\epsilon)}(\tau) = H \rightarrow M_\epsilon(\langle H \times \tau\rangle)$$

However, if we for example instantiate $\epsilon$ with the partial effect, any writes to the heap would be lost whenever an exception was thrown, which is not the expected semantics for using the heap. In general, using monad transformers to combine arbitrary effects may not always produce the desired semantics.

**io** It is important for computations to be able to interact with the outside world. This interaction is surprisingly challenging to represent though. One may expect that simply using an input stream and an output stream would suffice, but this fails to capture the fact that the inputs to the program can be affected by the outputs to the program. Instead, we expressed program interaction by defining a coinductive data type in terms of the various primitive interactive operations. To validate our formalization, we also designed a (non-terminating) process for evaluating an interactive program given an interactive environment. We then used this evaluation to form a quotient type of our coinductive data type in which all observably equivalent programs would be represented by the same value. This quotient type is our monad for the io effect. By construction, all primitive io operations can be expressed using this monad, as can infinitely recursive interactive programs. Also, there is a monad monomorphism from st-pure($ioheap$) by construction. This monadic construction is merely meant to prove soundness of io operations and semantics, and full details are beyond the scope of this paper.

### 6.4 Meaningful Soundness

From the definitions above, we can prove a very useful thing: our programs mean what one would expect them to mean. A total function from `int` to `int` is in fact a total function from $\mathbb{Z}_{2^{32}}$ to $\mathbb{Z}_{2^{32}}$. More formally, suppose every primitive type $\tau$ has a set representing its intended meaning: $[\![\tau]\!]$. Similarly each type constructor has an apprioprate corresponding function from sets to sets. In particular, the intended meaning of $(\tau_1, \ldots, \tau_n) \xrightarrow{\epsilon} \tau$ is $[\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \rightarrow M_\epsilon[\![\tau]\!]$. From this, we can give every type in our language an intended meaning. Then we claim the following:

***Theorem*** For every well-typed expression $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e : \tau|\epsilon$ (with no free type variables for simplicity), there is a corresponding mathematical function $[\![e]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_n]\!] \rightarrow M_\epsilon[\![\tau]\!]$.

***Corollary*** Our language has a sound type system and semantics.

## 7. Related work

The problems with arbitrary effects have been widely recognised, and there is a large body of work studying how to delimit the scope of effects. There have been many effect typing disciplines proposed. Early work is by Gifford and Lucassen (1986; 1988) which was later extended by Talpin (1993) and others (Talpin and Jouvelot 1994; Nielson et al. 1997). These systems are closely related since they describe polymorphic effect systems. Unfortunately, none of these have the same generality or rich effect structure that we describe. Moreover, to keep the systems decidable there are various constraints. For example, the system described by Nielson *et al.* (1997) requires the effects to form a complete lattice with meets and joins which we would be too weak to model our hierarchy (or user defined effects).

Tofte and others proposed a system for tracking region based memory allocation (Tofte and Birkedal 1998). Java contains a simple effect system where each method is labeled with the exceptions it might raise (Gosling et al. 1996). A system for finding uncaught exceptions was developed for ML by Pessaux *et al.* (1999). A more powerful system for tracking effects was developed by Benton (2007) who also studies the semantics of such effect systems (Benton et al. 2007). Variants of the ML value restriction is studied by Pessaux and Leroy (1993)

Tolmach (1998) describes an effect analysis for ML in terms of monads based on a subset of our effect hierarchy, namely total, partial, divergent and st. This is system is not polymorphic though and meant more for internal compiler analysis. In the context proof systems there has been work to show absence of observable side effects for object-oriented programming languages, for example by Naumann (2007).

Marino *et al.* recently produced a generic type-and-effect system (2009). This system uses privelege checking to describe analytical effect systems, and they provide a soundness proof for their type system. For example, an effect system could use try-catch statements to grant the `canThrow` privelege inside try blocks. `throw` statements are then only permitted when this privelege is present. Their system is very general and can express many properties. However, the specifications in their effect system have no semantics on their own. For example, according to their notion of soundness, it would also be sound for the effect system to have "+" grant the `canThrow` privelege to its arguments. Thus, one has to do an additional extensive proof to show that the effects in these systems actually correspond to an intended meaning. In essence, we define a coarse-grained denotational effect system whereas Marino *et al.* define a generic fine-grained analytical effect system.

Wadler and Thiemann showed the close relation between effect systems and monads (2003) and showed how any effect system can be translated to a monadic version – which is the case for our system too. There is a large body of work on the integration of subtyping and polymorphism. Early work was done by Mitchell (1991) which was later extended to a polymorphic setting (Smith 1991; Jones 1992). Pottier describes a more powerful simplification algorithm for subtype constraints (1996) (which is still incomplete).

## 8. Future work

There are still many items we wish to investigate further. Currently, our prototype implementation is very small and we are working on extending it to a more full-fledged language which would enable us to experiment with larger programs and study optimizations that exploit the effect information available. Another item we are working on is user-defined effects, enabling programmers to define custom effects by specifying monads along with various monad morphisms to integrate the custom effects into the effect hierarchy. The use of such effects corresponds closely to the definition of custom monads in Haskell and could be used to add effects such as "parser" and "environment". We are also interested in investigating how we can apply this work to existing ML and F# programs.

## References

Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI*, pages 15–26, 2007.

Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *Principles and practice of declarative programming*, pages 87–96, 2007.

David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LISP and functional programming*, pages 28–38, 1986.

James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. AW, 1996.

J.R. Hindley. The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.

Mark P. Jones. A theory of qualified types. In *ESOP*, volume 582 of *LNCS*, pages 287–306. Springer-Verlag, February 1992.

Mark P. Jones. From Hindley-Milner types to first-class structures. In *Proceedings of the Haskell Workshop*, June 1995.

David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *POPL*, pages 344–354, 1995.

David J. King and Philip Wadler. Combining monads. In *Glasgow Workshop on Functional Programming*, pages 134–143, 1993.

John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *ICFP*, pages 227–238, 1997.

Xavier Leroy. Polymorphism by name for references and continuations. In *POPL*, pages 220–231, 1993.

Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *POPL*, 1995.

J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL*, pages 47–57, 1988.

Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI*, pages 39–50, 2009.

Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:248–375, 1978.

J.C. Mitchell. Type inference with simple subtypes. *Journal of Functional Programming*, 1(3):245–286, 1991.

David A. Naumann. Observational purity and encapsulation. *Theoretical Compututer Science*, 376(3):205–224, 2007.

Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Analysis and Verification of Multiple-Agent Languages*, pages 141–171, 1997.

François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *POPL*, pages 276–290, 1999.

Simon L Peyton Jones and John Launchbury. State in Haskell. *Lisp and Symbolic Computation*, 8(4):293–341, 1995.

Francois Pottier. Simplifying subtyping constraints. In *ICFP*, pages 122–133, 1996.

John C. Reynolds. Using category theory to design implicit conversions and generic operators. *LNCS*, 94:211–258, 1980.

Geoffrey Seward Smith. *Polymorphic Type Inference for Languages with Overloading and Subtyping*. PhD thesis, Cornell University, August 1991.

Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and Computation*, 111(2):245–296, 1994.

J.P. Talpin. *Theoretical and practical aspects of type and effect inference*. PhD thesis, Ecole des Mines de Paris and University Paris VI, Paris, France, 1993.

Mads Tofte and Lars Birkedal. A region inference algorithm. *TOPLAS*, 20(4):724–767, 1998.

Andrew P. Tolmach. Optimizing ml using a hierarchy of monadic types. In *Types in Compilation*, pages 97–115, 1998.

Philip Wadler. Comprehending Monads. In *Mathematical Structures in Computer Science*, volume 2, pages 461–493, 1992.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. *Transactions on Computational Logic*, 4(1):1–32, 2003.