

Effectful programming

Type inference for effects with natural subtyping

(Draft, submitted to APLAS'10)

Ross Tate¹ and Daan Leijen²

¹ University of California, San Diego, rtate@cs.ucsd.edu

² 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:

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

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:

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

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 [1, 2], 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 [3] 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 [4], 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:

```
fib n = if n ≤ 0 then 1 else fib (n - 1) + 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:

```
fib n = if n ≤ 0 then do { print "hi"; return 1 }
      else do { x ← fib (n - 1);
              y ← fib (n - 2);
              return (x + y) }
```

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 an *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 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 F^{OX} . In essence one can see it as ML with controlled side effects, or as Haskell with strict semantics and implicit monads. In contrast to Haskell, 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 write function types as $(\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau$ to denote that a function takes arguments of type τ_1 to τ_n , and returns a value of type τ with a potential side effect ϵ . As apparent from the type, functions need to be fully applied and are not curried. This is to make it immediately apparent where side

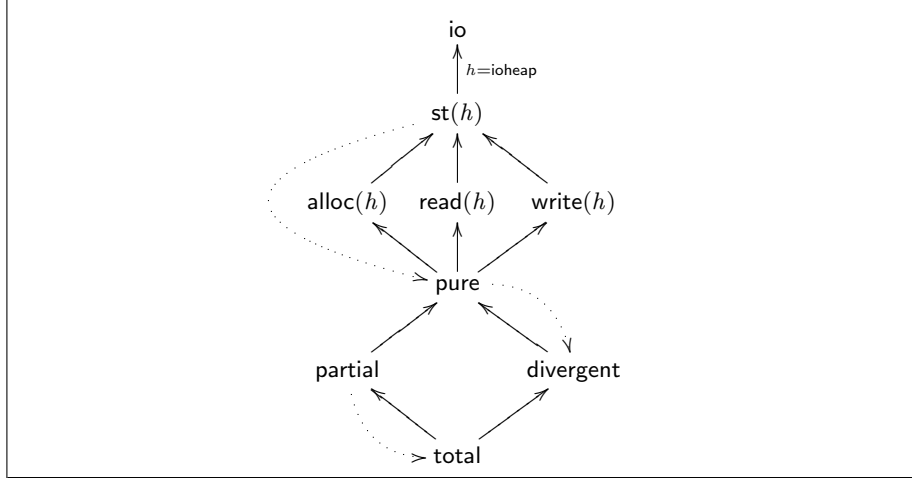


Fig. 1. The (simplified) effect hierarchy: an arrow $\epsilon_1 \rightarrow \epsilon_2$ implies that $\epsilon_1 \leq \epsilon_2$.

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 immediately apparent, as one writes either $f(x, y)$ or $(f(x))(y)$.

2.2 The basic effect hierarchy

The potential effects that we support form a semi-lattice under a sub-effect relation \leq , where we have for example that $\text{pure} \leq \text{io}$. In Figure 1 we show a simplified version of the effect hierarchy where each edge $\epsilon_1 \rightarrow \epsilon_2$ implies $\epsilon_1 \leq \epsilon_2$. 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.

total signifies the absence of any effects. In particular, total is a bottom element with $\text{total} \leq \epsilon$ for any effect ϵ . Functions with a total effect corresponds to total mathematical functions. The effects directly above total are partial and divergent , respectively representing expressions that possibly raise an exception or may not terminate. 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 pure effect is completely determined by the values of the arguments. 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 2.6.

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 $\text{alloc}(h)$ effect allocates in the heap h , $\text{read}(h)$ reads the heap, and $\text{write}(h)$ writes to the heap. The $\text{st}(h)$ effect is the join of those effects and can potentially allocate, read, and write to the heap – this type corresponds directly to the st monad in Haskell. 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. The dotted line implies that we can sometimes safely convert a stateful expression into a pure one as discussed in Section 2.3. The hierarchy of stateful effects is simplified here and we will refine it further in Section 2.3.

Finally, the `io` effect represents 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 `io` is not a sub effect of arbitrary stateful effects, only `st(h)` effects where the heap h is equal to the type constant `ioheap`.

2.3 State

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

```
fib(n : int) {
  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 ← e` binds a name to the result of an expression. The `repeat(n)` statement executes its body n times. A valid type for `fib` is:

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

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 quantified type variables as part of normal type inference.

Interestingly, the above function can be considered pure: 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 `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 [4]. We can in general not apply this rule automatically and the programmer needs to explicitly insert a `run` statement as follows:

$$fib(n : int) = \text{run}\{\dots\}$$

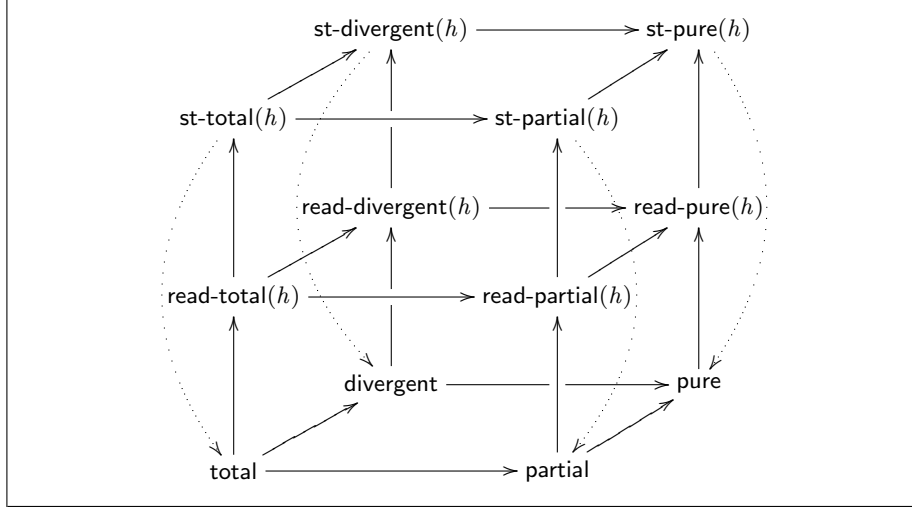


Fig. 2. The full relation between the basic effects and the expanded `read` and `st` effect.

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

$$fib : int \xrightarrow{\text{pure}} int$$

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

As said before, the diagram in Figure 1 is somewhat simplified and we can refine the above type even more. 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:

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

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 `read(h)` and `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 `fib` function will actually be `total`:

$$fib : int \xrightarrow{\text{total}} int$$

In Figure 2 the valid transitions induced by `run` are denoted by the dotted downward arrows. 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 [5]. Other examples of intricate stateful program that can be run as a pure programs are for example type unification [6], or the Garcia-Wachs algorithm [7].

2.4 Polymorphic effects

As already apparent from the previous examples, the type system supports full parametric polymorphism a la Hindley-Milner [1, 2]. 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:

$$\text{map}(f, xs) \{ \text{match } xs \text{ with} \\ \text{cons}(x, xs) \rightarrow \text{cons}(f(x), \text{map}(f, xs)) \\ \text{nil} \rightarrow \text{nil} \}$$

which we can type as:

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

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

2.5 No value restriction

Supporting both Hindley-Milner style polymorphic type inference and effects must be done carefully, since it is generally unsound in the presence of side effects. This problem occurs for example ML and lead to the introduction of the so-called value restriction in ML. Here is a small ML example showing the problem:

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

This program is unsound (and rejected in ML) since a list with a boolean element is assigned to reference *r* which is subsequently read back as a list of integers! 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 $\forall \alpha. \text{ref}(\text{list}(\alpha))$ 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. Unfortunately, this restriction also rejects many programs that are sound since there is no side effect involved, for example, $\text{let } \text{revlists} = \text{map } \text{rev} \text{ is rejected, while } \text{let } \text{revlists } xs = \text{map } \text{rev } xs \text{ is accepted.}$

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:

$$x \leftarrow 1/0; \text{ if } true \text{ then } 1 \text{ else } x$$

In the above program, we cannot replace the occurrence 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.

2.6 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 explicitly by the user. We can catch exceptions generally as:

$$catch : \forall \alpha \epsilon. (() \xrightarrow{\epsilon} \alpha, exn \xrightarrow{\epsilon} \alpha) \xrightarrow{\epsilon} \alpha \quad (\text{incorrect})$$

where the *catch* takes a function to evaluate and an exception handler that is executed if an exception was raised. Also note how parametric polymorphism naturally extends to polymorphic effects. However, if we specialize the type a bit more, we can transform partial expressions into a **total** one by using a **total** exception handler:

$$catchPartial : \forall \alpha \epsilon. (() \xrightarrow{\text{partial}} \alpha, () \xrightarrow{\epsilon} \alpha) \xrightarrow{\epsilon} \alpha$$

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:

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

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.

2.7 Partiality and non-termination

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.

3 The type system

The grammar of the types is given in Figure 3. Simple *monomorphic types* are written as τ and consist of type variables, type constructors, and functions. We usually write ϵ 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*(α). *Type schemes* σ have the form $\forall\alpha_1, \dots, \alpha_n. \tau$ where a type τ is quantified over the type variables α_1 to α_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 [8]. 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 simplification of constraints decidable. This property is easy to fulfill by ensuring that a function arrow is always (syntactically) fully applied.

Furthermore, the predicate $\tau_1 \leq \tau_2$ constrains a type τ_2 to be a subtype of τ_1 . The subtype relation $\tau_1 \leq \tau_2$ is assumed to form a semi-lattice 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 **partial** \leq **pure**, **read-total**(h) \leq **st-total**(h), and **read-total**(h) \leq **read-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.

3.1 Type rules

The grammar of expressions is defined in Figure 5 and the declarative type rules for expressions are defined in Figure 6. The declaration $\Gamma \vdash e : \tau \mid \epsilon$ states that expression e can be assigned the type τ with effect ϵ , assuming a type environment Γ . The type environment maps free variables to types. We write

Types,Effects	$\tau, \epsilon ::= \alpha$	(type variable)
	$ c(\tau_1, \dots, \tau_n)$	(type constructor)
	$ (\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau$	(function ($n \geq 0$))
Type schemes	$\sigma ::= \forall \bar{\alpha}. \tau$	(quantification)

Fig. 3. Types.

SUB-FUN	$\frac{\tau'_i \leq \tau_i \quad \tau \leq \tau' \quad \epsilon \leq \epsilon'}{(\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau \leq (\tau'_1, \dots, \tau'_n) \xrightarrow{\epsilon'} \tau'}$
SUB-TRANS	$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$
SUB-REFL	$\tau \leq \tau$

Fig. 4. Structural subtype rules.

environment extension as $(\Gamma, x : \sigma)$ which removes any previous bindings for x from Γ and adds the binding $x:\sigma$ to Γ . The inferred effect ϵ means that evaluation of the expression e can have a potential effect ϵ .

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 principal types and greatly simplifies type inference to be almost like type checking since only result types are inferred.

The rule for applications requires that the evaluation of the arguments and the function call have a common effect ϵ . Rule SUB and LIFT can always be used to lift expressions with a different effect into a common effect ϵ (since the effects form a semi-lattice). The next two rules deal with binding names to expressions. The LET rule allows names to be bound to a polymorphic type σ , 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 τ .

Rule GEN is the usual generalization rule where a type can be generalized over variables $\bar{\alpha}$ as long as $\bar{\alpha}$ 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 2.5 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 τ , we can also derive a type τ' if that is a subtype of τ , i.e. $\tau \leq \tau'$.

$e ::= x$	(variable)
$e(e_1, \dots, e_n)$	(function application)
$\lambda(x_1 : \tau_1, \dots, x_n : \tau_n) \rightarrow e$	(function definition)
$x = e_1; e_2$	(let binding)
$x \leftarrow e_1; e_2$	(effect binding)
run e	(run isolated state)

Fig. 5. Expression grammar.

VAR	$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \mid \mathbf{total}}$
FUN	$\frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau \mid \epsilon}{\Gamma \vdash \lambda(x_1 : \tau_1, \dots, x_n : \tau_n) \rightarrow e : (\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau \mid \mathbf{total}}$
APP	$\frac{\Gamma \vdash e : (\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau \mid \epsilon \quad \Gamma \vdash e_i : \tau_i \mid \epsilon}{\Gamma \vdash e(e_1, \dots, e_n) : \tau \mid \epsilon}$
LET	$\frac{\Gamma \vdash e_1 : \sigma_1 \mid \mathbf{total} \quad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash x = e_1; e_2 : \tau_2 \mid \epsilon}$
BIND	$\frac{\Gamma \vdash e_1 : \tau_1 \mid \epsilon \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash x \leftarrow e_1; e_2 : \tau_2 \mid \epsilon}$
GEN	$\frac{\Gamma \vdash e : \tau \mid \mathbf{total} \quad \bar{\alpha} \notin ftv(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}. \tau \mid \mathbf{total}}$
INST	$\frac{\Gamma \vdash e : \forall \bar{\alpha}. \tau \mid \epsilon}{\Gamma \vdash e : [\bar{\alpha} := \bar{\tau}] \tau \mid \epsilon}$
SUB	$\frac{\Gamma \vdash e : \tau_1 \mid \epsilon \quad \tau_1 \leq \tau_2}{\Gamma \vdash e : \tau_2 \mid \epsilon}$
LIFT	$\frac{\Gamma \vdash e : \tau \mid \epsilon_1 \quad \epsilon_1 \leq \epsilon_2}{\Gamma \vdash e : \tau \mid \epsilon_2}$
RUN-PURE	$\frac{\Gamma \vdash e : \tau \mid \mathbf{st-pure}(h) \quad h \notin ftv(\tau, \Gamma)}{\Gamma \vdash \mathbf{run} \ e : \tau \mid \mathbf{pure}}$

Fig. 6. Type rules.

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 ϵ_1 , we can also derive a ‘worse’ effect ϵ_2 if these are in a subtype relation: $\epsilon_1 \leq \epsilon_2$. 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.

Since all function parameters are annotated with their type, type checking is straightforward and there is no need for unification for example. The rules can be made syntax directed in a straightforward way: instantiation, subtyping, and

lifting is only performed on the arguments in an application rule to match the expected types, and generalization is only applied to let-bound expressions. A key property that makes the system decidable is that with our restrictive subtype system, all monomorphic types and effects have necessary joins, i.e. there exists a join whenever there exists some common supertype.

3.2 Divergence

We assume that primitive effect operations like throwing an exception, reference creation and assignment etc. are defined as primitive functions, similarly to stateful operations in Haskell. One effect that cannot be described that way is divergence. Instead divergence is introduced by recursion. We use the standard `fix` primitive to introduce recursion. Of course, if we know the recursion is always terminating, we do not require the result to be in the divergent effect. We therefore introduce two rules: one for terminating recursive functions, and one for potentially divergent ones:

$$\text{FIX-TERM} \quad \frac{\Gamma \vdash e : \tau \xrightarrow{\epsilon} \tau \mid \text{total} \quad e \text{ is terminating}}{\Gamma \vdash \text{fix } e : \tau \mid \epsilon}$$

$$\text{FIX-DIV} \quad \frac{\Gamma \vdash e : \tau \xrightarrow{\epsilon} \tau \mid \text{total} \quad \text{divergent} \leq \epsilon}{\Gamma \vdash \text{fix } e : \tau \mid \epsilon}$$

The first rule can only apply when a recursive expression can be shown to be terminating. When this predicate is true of course depends on the particular termination analysis that one might implement. As described in Section 2.7, in our current system this predicate is only true for structurally recursive functions over inductive datatypes. The next rule `FIX-DIV` applies in all other cases but it requires that the result effect is a sub-effect of `divergent` ensuring that any potentially divergent function has a divergent effect (like `pure`, or `st-divergent` for example).

4 A type-directed monadic translation

Figure 7 defines a type-directed translation from the basic type system to the predicative fragment of System F. We assume the existence of primitive monadic operations, and for each effect ϵ we assume the definition of a monad M_ϵ , together with the following primitive monad operations which should satisfy the monad laws:

$$\begin{aligned} \gg_{\epsilon} &: \forall \alpha \beta. M_\epsilon(\alpha) \rightarrow (\alpha \rightarrow M_\epsilon(\beta)) \rightarrow M_\epsilon(\beta) && \text{(bind)} \\ \text{unit}_\epsilon &: \forall \alpha. \alpha \rightarrow M_\epsilon(\alpha) && \text{(return)} \end{aligned}$$

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

$$\text{run}_{\text{pure}} : \forall \alpha. (\forall h. M_{\text{st-pure}(h)}(\alpha)) \rightarrow M_{\text{pure}}(\alpha) \quad \text{(runST)}$$

We treat the `total` monad specially though and treat $M_{\text{total}}(\sigma)$ as a type synonym for σ , i.e. `total` is the identity monad. As such, the bind operation for `total` is just

VAR	$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma \mid \mathbf{total} \rightsquigarrow x}$
FUN	$\frac{\Gamma, x_1 : \tau_1, \dots, x_n : \tau_n \vdash e : \tau \mid \epsilon \rightsquigarrow \mathbf{e}}{\Gamma \vdash \lambda(x_1, \dots, x_n) \rightarrow e : (x : \tau_1, \dots, x : \tau_n) \xrightarrow{\epsilon} \tau \mid \mathbf{total} \rightsquigarrow \lambda(x_1 : \tau_1, \dots, x_n : \tau_n). \mathbf{e}}$
APP	$\frac{\Gamma \vdash e : (\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau \mid \epsilon \rightsquigarrow \mathbf{e} \quad \Gamma \vdash e_i : \tau_i \mid \epsilon \rightsquigarrow \mathbf{e}_i}{\Gamma \vdash e(e_1, \dots, e_n) : \tau \mid \epsilon \rightsquigarrow \mathbf{e} \quad \mathbf{e} \gg_{\epsilon} \lambda(x : (\tau_1, \dots, \tau_n) \xrightarrow{\epsilon} \tau). \quad \mathbf{e}_1 \gg_{\epsilon} \lambda(x_1 : \tau_1). \quad \dots \quad \mathbf{e}_n \gg_{\epsilon} \lambda(x_n : \tau_n). x(x_1, \dots, x_n)}$
LET	$\frac{\Gamma \vdash e_1 : \sigma_1 \mid \mathbf{total} \rightsquigarrow \mathbf{e}_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash x = e_1; e_2 : \tau_2 \mid \epsilon \rightsquigarrow (\lambda(x : \sigma_1). \mathbf{e}_2) \mathbf{e}_1}$
BIND	$\frac{\Gamma \vdash e_1 : \tau_1 \mid \epsilon \rightsquigarrow \mathbf{e}_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathbf{e}_2}{\Gamma \vdash x \leftarrow e_1; e_2 : \tau_2 \mid \epsilon \rightsquigarrow \mathbf{e}_1 \gg_{\epsilon} \lambda(x : \tau_1). \mathbf{e}_2}$
GEN	$\frac{\Gamma \vdash e : \tau \mid \mathbf{total} \rightsquigarrow \mathbf{e} \quad \bar{\alpha} \not\in \mathit{ftv}(\Gamma)}{\Gamma \vdash e : \forall \bar{\alpha}. \tau \mid \mathbf{total} \rightsquigarrow \Lambda(\alpha_1, \dots, \alpha_n). \mathbf{e}}$
INST	$\frac{\Gamma \vdash e : \forall \bar{\alpha}. \tau \mid \epsilon \rightsquigarrow \mathbf{e} \quad \bar{\tau} = \tau_1, \dots, \tau_n}{\Gamma \vdash e : [\bar{\alpha} := \bar{\tau}] \tau \mid \epsilon \rightsquigarrow \mathbf{e}[\tau_1, \dots, \tau_n]}$
SUB	$\frac{\Gamma \vdash e : \tau_1 \mid \epsilon \rightsquigarrow \mathbf{e} \quad \tau_1 \leq \tau_2 \rightsquigarrow \mathbf{f}}{\Gamma \vdash e : \tau_2 \mid \epsilon \rightsquigarrow \mathbf{e} \gg_{\epsilon} (\mathit{unit}_{\epsilon} \circ \mathbf{f})}$
LIFT	$\frac{\Gamma \vdash e : \tau \mid \epsilon_1 \rightsquigarrow \mathbf{e} \quad \epsilon_1 \leq \epsilon_2 \rightsquigarrow \mathbf{f}}{\Gamma \vdash e : \tau \mid \epsilon_2 \rightsquigarrow \mathbf{f}(\mathbf{e})}$
RUN-PURE	$\frac{\Gamma \vdash e : \tau \mid \mathbf{st-pure}(h) \rightsquigarrow \mathbf{e} \quad h \notin \mathit{ftv}(\tau, \Gamma)}{\Gamma \vdash \mathit{run} \ e : \tau \mid \mathbf{pure} \rightsquigarrow \mathit{run}_{\mathbf{pure}}(\Lambda h. \mathbf{e})}$

Fig. 7. Monadic type-directed translation.

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 $\tau_1 \rightarrow M_{\epsilon}(\tau_2)$ when necessary.

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

$$\text{SUB-TRANS} \quad \frac{\tau_1 \leq \tau_2 \rightsquigarrow \mathbf{f}_1 \quad \tau_2 \leq \tau_3 \rightsquigarrow \mathbf{f}_2}{\tau_1 \leq \tau_3 \rightsquigarrow \mathbf{f}_2 \circ \mathbf{f}_1}$$

Using these primitives, the monadic type directed translation in Figure 7 is straightforward and follows exactly the structure of the type derivation. Each rule $\Gamma \vdash e : \sigma \mid \epsilon \rightsquigarrow \mathbf{e}$ states that if we derive a type σ with effect ϵ for an

expression e under the environment Γ , we can also derive a corresponding well-typed System-F expression e with type $M_e(\sigma)$.

Well-typedness of the resulting (predicative) System F expression is straightforward to prove by induction over the type rules. As a consequence, since System F is sound, our type system is sound under the assumption that our primitive monad operations are sound. Fortunately, the monads we use are well established and have already been demonstrated sound by many others [9–12] including interactive input- and output [10], and including our run primitive [13, 4], and thus our type system is in fact sound.

Apart from divergence, we can actually model most effects directly in System F. For example, using an ML-like syntax, the partial monad can be defined as:

$$\mathbf{type} \ M_{\text{partial}}(\tau) \quad = \text{Ok } \tau \mid \text{Error}$$

$$\mathit{unit}_{\text{partial}} \ x \quad = \text{Ok } x$$

$$(\text{Ok } x) \ggg_{\text{partial}} f \quad = f \ x$$

$$\text{Error} \ggg_{\text{partial}} f \quad = \text{Error}$$

and similarly for all the stateful effects [14, 13]. By construction, all of the effects that can be encoded in System F are of course sound.

5 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 [15, 16] which was later extended by Talpin [17] and others [18, 19]. 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.* [19] 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 [20]. Java contains a simple effect system where each method is labeled with the exceptions it might raise [21]. A system for finding uncaught exceptions was developed for ML by Pessaux *et al.* [22]. A more powerful system for tracking effects was developed by Benton [3] who also studies the semantics of such effect systems [23]. Variants of the ML value restriction is studied by Pessaux and Leroy [24]

Tolmach [25] 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 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 [26].

Marino *et al.* recently produced a generic type-and-effect system [27]. This system uses privilege 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` privilege inside try blocks. `throw` statements are then only permitted when this privilege 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` privilege 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 [11] and showed how any effect system can be translated to a monadic version – which is the case for our system too. Launchbury and Sabry [13] show how the state can be safely encapsulated using `runST` in a state monad.

6 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. We are also interested in investigating how we can apply this work to existing ML and F# programs. Type inference becomes much more challenging in that setting since function parameters are not annotated and must be inferred. In general, type inference with subtyping and parametric polymorphism is undecidable and presents a challenge even with our restricted subtype relation, and we describe a preliminary solution based on qualified types in a technical report [28].

References

1. Hindley, J.: The principal type scheme of an object in combinatory logic. *Transactions of the American Mathematical Society* **146** (1969) 29–60
2. Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Sciences* **17** (1978) 248–375
3. Benton, N., Buchlovsky, P.: Semantics of an effect analysis for exceptions. In: TLDI. (2007) 15–26
4. Peyton Jones, S.L., Launchbury, J.: State in Haskell. *Lisp and Symbolic Computation* **8**(4) (1995) 293–341
5. King, D.J., Launchbury, J.: Structuring depth-first search algorithms in Haskell. In: POPL. (1995) 344–354
6. Peyton Jones, S., Vytiniotis, D., Weirich, S., Shields, M.: Practical type inference for arbitrary-rank types. *Journal of Functional Programming* **17**(1) (2007) 1–82
7. Filliâtre, J.C.: A functional implementation of the Garsia–Wachs algorithm: (functional pearl). In: ML. (2008) 91–96

8. Jones, M.P.: From Hindley-Milner types to first-class structures. In: Proceedings of the Haskell Workshop. (1995)
9. Braner, T.: A simple adequate categorical model for PCF. In: In Proceedings of Third International Conference on Typed Lambda Calculi and Applications, Springer-Verlag (1997) 82–98
10. Moggi, E.: Notions of computation and monads. *Information and Computation* **93** (1989) 55–92
11. Wadler, P., Thiemann, P.: The marriage of effects and monads. *Transactions on Computational Logic* **4**(1) (2003) 1–32
12. Mitchell, J.C.: *Foundations for Programming Languages*. The MIT press (1996)
13. Launchbury, J., Sabry, A.: Monadic state: Axiomatization and type safety. In: ICFP. (1997) 227–238
14. Wadler, P.: The essence of functional programming. In: 19'th Symposium on Principles of Programming Languages, Albuquerque, New Mexico, ACM press (1992) 1–14
15. Gifford, D.K., Lucassen, J.M.: Integrating functional and imperative programming. In: *LISP and functional programming*. (1986) 28–38
16. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: *POPL*. (1988) 47–57
17. Talpin, J.: Theoretical and practical aspects of type and effect inference. PhD thesis, Ecole des Mines de Paris and University Paris VI, Paris, France (1993)
18. Talpin, J.P., Jouvelot, P.: The type and effect discipline. *Information and Computation* **111**(2) (1994) 245–296
19. Nielson, H.R., Nielson, F., Amtoft, T.: Polymorphic subtyping for effect analysis: The static semantics. In: *Analysis and Verification of Multiple-Agent Languages*. (1997) 141–171
20. Tofte, M., Birkedal, L.: A region inference algorithm. *TOPLAS* **20**(4) (1998) 724–767
21. Gosling, J., Joy, B., Steele, G.: *The Java Language Specification*. AW (1996)
22. Pessaux, F., Leroy, X.: Type-based analysis of uncaught exceptions. In: *POPL*. (1999) 276–290
23. Benton, N., Kennedy, A., Beringer, L., Hofmann, M.: Relational semantics for effect-based program transformations with dynamic allocation. In: *Principles and practice of declarative programming*. (2007) 87–96
24. Leroy, X.: Polymorphism by name for references and continuations. In: *POPL*. (1993) 220–231
25. Tolmach, A.P.: Optimizing ml using a hierarchy of monadic types. In: *Types in Compilation*. (1998) 97–115
26. Naumann, D.A.: Observational purity and encapsulation. *Theoretical Computer Science* **376**(3) (2007) 205–224
27. Marino, D., Millstein, T.: A generic type-and-effect system. In: *TLDI*. (2009) 39–50
28. Tate, R., Leijen, D.: Convenient explicit effects using type inference with subeffects. Technical Report MSR-TR-2010-XX, Microsoft Research (2010)