Koka: Programming with Row-polymorphic Effect Types

Daan Leijen
Microsoft Research, MSR-TR-2013-79 (28 Aug 2013)
daan@microsoft.com

Note: *This is an updated article: a previous version of this article contained a wrong lemma and corresponding mistakes in various proofs of Section 5.*

Abstract

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. 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. A novel feature is that we support polymorphic effects through row-polymorphism using duplicate labels. Moreover, we show that our effects are not just syntactic labels but have a deep semantic connection to the program. For example, if an expression can be typed without an *exn* effect, then it will never throw an unhandled exception. Similar to Haskell's $runST$ we show how we can safely encapsulate stateful operations. Through the state effect, we can also safely combine state with let- polymorphism without needing either imperative type variables or a syntactic value restriction. Finally,our system is implemented fully in a new language called Koka[1] and has been used successfully on various small to medium-sized sample programs ranging from a Markdown processor to a tier-splitted chat application.

1. Introduction

We propose a programming model where effects are a part of the type signature of a function. Currently, types only tell us something about the input and output value of a function but say nothing about all *other* behaviors; for example, if the function writes to the console or can throw an exception. In our system, the squaring function:

```
function sqr(x : int) { x * x }
```

will get the type:

$$sqr \;:\; int \rightarrow total\ int$$

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

```
function sqr(x : int) { print(x); x * x }
```

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

$$sqr \;:\; int \rightarrow io\ int$$

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. 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.

There have been many proposals for effects systems in the past [7, 18, 30, 36, 22, 2, 19, 32, 27]. However, many such systems suffer from being syntactical in nature (i.e. effects are just labels), or by being quite restricted, for example being monomorphic or applied to a very specific set of effects. Some of the more general systems suffer from having complicated effect types, especially in a polymorphic setting that generally requires sub- effect constraints.

This paper has been long in the works and brings together many pieces of the effect puzzle: ranging from effect types as rows with duplicate labels, the semantics of effect types, and practical experience with effect inference. In particular, we make the following contributions:

- We describe a novel effect system based on row polymorphism which allows *duplicated* effects. It turns out that this is essential to provide natural types to effect elimination forms, like catching exceptions.
- The effect types are not just syntactic labels but they have a deep semantic connection to the program (Section 7). For example, we can prove that if an expression that can be typed without an *exn* effect, then it will never throws an unhandled exception; or if an expressions can be typed without a *div* effect, then it will always terminate.
- The interaction between polymorphism and mutable state is fraught with danger. We show that by modeling state as an effect we can safely combine mutability with let-polymorphism without needing either *imperative type variables*, nor a syntactic value restriction.

  Moreover, we can safely encapsulate local state operations and we prove that such encapsulation is sound where no references or stateful behavior can escape the encapsulation scope.

  The interaction between divergence and higher-order mutable state is also tricky. Again, we show how explicit heap effects allow us to safely infer whether stateful operations may diverge.
- Having to keep track of effects manually would be a large burden: we describe a sound and complete type inference system that automatically infers the principal effect and type of any expression, and automatically promotes effects when necessary. (Section 4).
- We have an extensive experience with the type system within the Koka language. The Koka language fully implements the effect types as described in this paper and we have used it successfully in various small to medium sized code examples ranging from a fully compliant Markdown text processor to a tier-splitted chat application.

[1] Koka means ‘effect’ or ‘effective’ in Japanese.

2. Overview

Types tell us about the behavior of functions. For example, if suppose we have the type of a function *foo* in ML:

$$\mathit{foo}\ :\ \mathit{int} \to \mathit{int}$$

we know that *foo* is well defined on inputs of type int and returns values of type int. But that that is only part of the story, the type tells us nothing about all *other* behaviors of foo. For example, we do not know if this is truly a function in the mathematical sense, returning the same result for same inputs. Or if it accesses the file system perhaps, or throws exceptions, or never returns a result at all. Even 'pure' functional languages like Haskell do not fare much better at this. Suppose the Haskell function *foo* has type:

$$\mathit{foo}\ ::\ \mathit{Int} \to \mathit{Int}$$

Even though we know now there is no arbitrary side-effect, we still do not know whether this function will terminate or raise exceptions. Due to laziness, we do not even know if the result itself, when demanded, will raise an exception or diverge; i.e. even a simple transformation like $x * 0$ to 0 is not possible under Haskell's notion of purity. In essence, in both ML and Haskell the types are not precise enough to describe many aspects of the static behavior of a program. In the Haskell case, the real type is more like $(Int_{\bot} \to Int_{\bot})_{\bot}$ while the type signature of the ML program should really include that any kind of side-effect might happen.

We have been doing it wrong all this time! We believe it is essential for types to include potential behaviors like divergence, exceptions, or stateful- ness. Being able to reason about these aspects is crucial in many domains, including safe parallel execution, optimization, query embedding, tier- splitting, etc.

2.1. Effect types

To address the previous problems, we took a fresh look at programming with side-effects, and developed a new language called Koka [15]. Like ML, Koka has 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 $(\tau_1, \ldots, \tau_n) \to \epsilon\ \tau$ 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 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. Basic effects

The effects in our system are extensible, but the basic effects defined in Koka are $total$, exn, div, $ndet$, $alloc\langle h\rangle$, $read\langle h\rangle$, $write\langle h\rangle$, and io. Of course $total$ is not really an effect but signifies the absence of any effect and is assigned to pure mathematical functions. When a function can raise an exception, it will get the exn effect. Potential divergence or non-termination is signified by the div effect. Currently, Koka uses a simple termination analysis based on inductive data types to assign this effect to recursive functions. Non-deterministic functions get the $ndet$ effect. The effects $alloc\langle h\rangle$, $read\langle h\rangle$ and $write\langle h\rangle$ are used for stateful functions over a heap h. Finally io is used for functions that do any input/output operations.

Here are some type signatures of common functions in Koka:

$$\begin{array}{lll}
\mathit{random} & : & () \to \mathit{ndet}\ \mathit{double} \\
\mathit{print} & : & \mathit{string} \to \mathit{io}\ () \\
\mathit{error} & : & \forall\alpha.\ \mathit{string} \to \mathit{exn}\ a \\
(:=) & : & \forall\alpha.\ (\mathit{ref}\langle h, a\rangle,\ a) \to \mathit{write}\langle h\rangle\ ()
\end{array}$$

Note that we use angled brackets to denote type application as usual in languages like C# or Scala. We also use angled brackets to denote a *row* of effects. For example, the program:

$$\textsf{function}\ \mathit{sqr}(\ x\ :\ \mathit{int}\ )\ \{\mathit{error}(\texttt{"hi"});\ \mathit{sqr}(x);\ x * x\ \}$$

will get the type

$$\mathit{sqr}\ :\ \mathit{int} \to \langle \mathit{exn}, \mathit{div}\rangle\ \mathit{int}$$

where we combined the two basic effects exn and div into a row of effects $\langle exn,\ div\rangle$. The combination of the exception and divergence effect corresponds exactly to Haskell's notion of purity, and we call this effect $pure$. Other useful type aliases include:

$$\begin{array}{lll}
\textsf{alias}\ \mathit{total} & = & \langle\rangle \\
\textsf{alias}\ \mathit{pure} & = & \langle \mathit{exn},\ \mathit{div}\rangle \\
\textsf{alias}\ \mathit{st}\langle h\rangle & = & \langle \mathit{alloc}\langle h\rangle,\ \mathit{read}\langle h\rangle,\ \mathit{write}\langle h\rangle\rangle \\
\textsf{alias}\ \mathit{io} & = & \langle \mathit{st}\langle \mathit{ioheap}\rangle,\ \mathit{pure},\ \mathit{ndet}\rangle
\end{array}$$

This hierarchy is clearly inspired by Haskell's standard monads and we use this as a starting point for more refined effects which we hope to explore in Koka. For example, blocking, client/server side effects, reversable operations, etc.

2.3. Polymorphic effects

Often, the effect of a function is determined by the effects of functions passed to it. For example, the map function which maps a function over all elements of a list will have the type:

$$\mathit{map}\ :\ \forall\alpha\beta\mu.\ (\mathit{list}\langle\alpha\rangle,\ \beta \!\to \mu\ \beta) \to \mu\ \mathit{list}\langle\beta\rangle$$

where the effect of the map function itself is completely determined by the effect of its argument. In this case, a simple and obvious type is assigned to map, but one can quickly create more complex examples where the type may not be obvious at first. Consider the following program:

$$\textsf{function}\ \mathit{foo}(f, g)\ \{\ f();\ g();\ \mathit{error}(\text{"}\mathit{hi}\text{"})\ \}$$

Clearly, the effect of foo is a combination of the effects of f and g, and the exn effect. One possible design choice is to have a $\cup$ operation on effect types, and write the type of foo as:

$$\forall\mu_1\mu_2.\ (() \to \mu_1\ (),\ () \to \mu_2\ ()) \to (\mu_1 \cup \mu_2 \cup \mathit{exn})\ ()$$

Unfortunately, this quickly gets us in trouble during type inference: unification can lead to constraints of the form $\mu_1 \cup\ \mu_2 \sim \mu_3 \cup\ \mu_4$ which cannot be solved uniquely and must become part of the type language. Another design choice is to introduce subtyping over effects and write the type of foo as:

$$\begin{array}{l}
\forall\mu_1\mu_2\mu_3.\ (\mu_1 \le \mu_3,\ \mu_2 \le \mu_3,\ \langle \mathit{exn}\rangle \le \mu_3)\ \Rightarrow \\
\qquad\qquad (() \to \mu_1\ (),\ () \to \mu_2\ ()) \to \mu_3\ ()
\end{array}$$

This is the choice made in an earlier version of Koka described as a technical report [32]. However, in our experience with that system in practice we felt the constraints often became quite complex and the combination of polymorphism with subtyping can make type inference undecidable.

The approach we advocate in this paper and which is adopted by Koka is the use of row-polymorphism on effects. Row polymorphism is well understood and used for many inference systems for record calculi [6, 14, 26, 29, 28, 17]. We use the notation $\langle l \mid \mu\rangle$ to extend an effect row μ with an effect constant l. Rows can now have two forms, either a *closed* effect $\langle exn, div\rangle$, or an *open* effect ending in an effect variable $\langle exn, div \mid \mu\rangle$. Using an open effect, our system infers the following type for *foo*:

$$foo \;:\; \forall\mu.\ (() \to \langle exn \mid \mu\rangle\ (),\ () \to \langle exn \mid \mu\rangle\ ()) \to \langle exn \mid \mu\rangle\ ()$$

The reader may worry at this point that the row polymorphic type is more restrictive than the earlier type using subtype constraints: indeed, the row polymorphic type requires that each function argument now has the same effect $\langle exn \mid \mu\rangle$. However, in a calling context $foo(f, g)$ our system ensures that we always infer a polymorphic open effect for each expression f and g. For example, $f \,:\, () \to \langle exn \mid \mu_1\rangle\ ()$ and $g \,:\, () \to \langle div \mid \mu_2\rangle\ ()$. This allows the types $\langle exn \mid \mu_1\rangle$ and $\langle div \mid \mu_2\rangle$ to unify into a common type $\langle exn, div \mid \mu_3\rangle$ such that they can be applied to *foo*, resulting in an inferred effect $\langle exn, div \mid \mu_3\rangle$ for $foo(f, g)$.

2.4. Duplicate effects

Our effect rows differ in an important way from the usual approaches in that effect labels can be duplicated, i.e. $\langle exn, exn\rangle \not\equiv \langle exn\rangle$ **(1)**. This was first described by Leijen [14] where this was used to enable scoped labels in record types. Enabling duplicate labels is crucial for our approach. First of all, it enables principal unification without needing extra constraints and secondly, it enables us to give precise types to effect elimination forms (like catching exceptions).

In particular, during unification we may end up with constraints of the form $\langle exn \mid \mu\rangle \sim \langle exn\rangle$. With regular row-polymorphism, such constraint can have multiple solutions, namely $\mu = \langle\rangle$ or $\mu = \langle exn\rangle$. This was first observed by Wand [37] in the context of records. Usually, this problem is fixed by either introducing *lacks* constraints [6] or polymorphic presence and absence flags on each label [25] (as used by Lindley and Cheney [17] for an effect system in the context of database queries). With rows allowing duplicate labels, we avoid additional machinery since in our case $\mu = \langle\rangle$ is the only solution to the above constraint (due to (1)).

Moreover, duplicate labels make it easy to give types to effect elimination forms. For example, catching effects removes the exn effect:

$$catch \;:\; \forall\alpha\mu.\ (() \to \langle exn \mid \mu\rangle\ \alpha,\ exception \to \mu\ \alpha) \to \mu\ \alpha$$

Here we assume that $catch$ takes two functions, the action and the exception handler that takes as an argument the thrown $exception$. Here, the exn effect of the action is discarded in the final effect μ since all exceptions are handled by the handler. But of course, the handler can itself throw an exception and have an exn effect itself. In that case μ will unify with a type of the form $\langle exn|\mu'\rangle$ giving action the effect $\langle exn|exn|\mu'\rangle$ where exn occurs duplicated, which gives us exactly the right behavior. Note that with *lacks* constraints we would not be able to type this example because there would be a $exn \not\in \mu$ constraint. We can type this example using flags but the type would arguably be more complex with a polymorphic presence/absence flag on the exn label in the result effect, something like:

$$\begin{aligned} catch \;:\; \forall\mu\alpha\varphi.\ (() \to \langle exn_\bullet | \mu\rangle\ \alpha,\ exception \to \langle exn_\varphi | \mu\rangle\ \alpha) \\ \to \langle exn_\varphi | \mu\rangle\ \alpha \end{aligned}$$

There is one situation where an approach with flags is more expressive though: with flags one can state specifically that a certain effect must be absent. This is used for example in the effect system by Lindley and Cheney [17] to enforce that database queries never have the *wild* effect (*io*). In our setting we can only enforce absence of an effect by explicitly listing a closed row of the allowed effects which is less modular. In our current experience this has not yet proven to be a problem in practice though.

2.5. Heap effects

One of the most useful side-effects is of course mutable state. Here is an example where we give a linear version of the fibonacci function using imperative updates:

```
function fib( n : int )
{
  val x = ref(0)
  val y = ref(1)
  repeat(n) {
    val y0 = !y
    y := !x+!y
    x := y0
  }
  !x
}
```

Here x and y are bound to freshly allocated references of type $ref\langle h, int\rangle$. The operator (!) dereferences a reference while the operator (:=) is used for assignment to references.

Due to the reading and writing of x and y of type $ref\langle h, int\rangle$, the effect inferred for the body of the function is $st\langle h\rangle$ for some heap h. As such, a valid type for *fib* would be:

$$fib \;:\; \forall h.\ int \to st\langle h\rangle\ int$$

However, we can of course consider the function *fib* to be total: 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 remove the effect $st\langle h\rangle$ 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 corresponds directly to the use of the higher-ranked $runST$ function in Haskell [24] (which we will call just run):

$$run \;:\; \forall\mu\alpha.\ (\forall h.\ () \to \langle st\langle h\rangle \mid \mu\rangle\ \alpha) \to \mu\ \alpha$$

Koka will automatically insert a run wrapper at generalization points if it can be applied, and infers a total type for the above fibonacci function:

$$fib \;:\; int \to total\ int$$

Again, using row polymorphism is quite natural to express in the type of run where the $st\langle h\rangle$ effect can be dismissed.

One complex example from a type inference perspective where we applied Koka, is the Garsia-Wachs algorithm as described by Filliâtre [5]. The given algorithm was originally written in ML and uses updateable references in the leaf nodes of the trees to achieve efficiency comparable to the reference C implementation. However, Filliâtre remarks that these side effects are local and not observable to any caller. We implemented Filliâtre's algorithm in Koka and our system correctly inferred that the state effect can be discarded and assigned a pure effect to the Garsia-Wachs algorithm [15].

2.6. Heap safety

Combining polymorphism and imperative state is fraught with difficulty and requires great care. In particular, *let*-polymorphism may lead to unsoundness if references can be given a polymorphic type. A classical example from ML is:

$$\mathsf{let}\ r \;=\; ref\ [\,]\ \mathsf{in}\ (r \;:=\; [true],\ !r + 1)$$

Here, we let bind r to a reference with type $\forall\alpha.\ ref\langle list\langle\alpha\rangle\rangle$. The problem is that this type can instantiate later to both a reference to an integer list and a boolean list. Intuitively, the problem is that the first binding of r generalized over type variables that are actually free in the heap. The ML language considered many solutions to prevent this from happening, ranging from imperative type variables [33] to the current syntactic value restriction, where only value expressions can be generalized.

In our system, no such tricks are necessary. Using the effect types, we restrict generalization to expressions that are total, and we reject the ML example since we will not generalize over the type of r since it has an $alloc\langle h\rangle$ effect. We prove in Section 6.1 that our approach is semantically sound. In contrast to the value restriction, we can still generalize over any expression that is not stateful regardless of its syntactic form.

The addition of run adds further requirements where we must ensure that encapsulated stateful computations truly behave like a pure function and do not 'leak' the state. For example, it would be unsound to let a reference escape its encapsulation:

```
run( function(){ ref(1) })
```

or to encapsulate a computation where its effects can still be observed:

```
function wrong() {
  val r = ref(1)
  function foo() { run( function(){ r := !r + 1 }) }  (looks pure)
  foo()
  !r    (expecting 1 but getting 2)
}
```

We prove in Section 6.2 that well-typed terms never exhibit such behavior. To our knowledge we are the first to prove this formally for a strict language in combination with exceptions and divergence. A similar result is by Launchbury and Sabry [13] where they prove heap safety of the Haskell's ST monad in the context of a lazy store with lazy evaluation.

2.7. Divergence

Koka uses a simple termination checker (based on [1]) to assign the divergence effect to potentially non-terminating functions. To do this safely, Koka has three kinds of data types, inductive, co-inductive, and arbitrary recursive data types. In particular, we restrict (co)inductive data types such that the type itself cannot occur in a negative position. Any function that matches on an arbitrary recursive data type is assumed to be potentially divergent since one can encode the Y combinator using such data type and write a non-terminating function that is not syntactically recursive.

Recursively defined functions should of course include the divergence effect in general. However, if the termination checker finds that each recursive call decreases the size of an inductive data type (or is productive for a co- inductive data type), then we do not assign the divergent effect. The current analysis is quite limited and syntactically fragile but seems to work well enough in practice (Section 2.8). For our purpose, we prefer a predictable analysis with clear rules.

However, in combination with higher-order mutable state, we can still define functions that are not syntactically recursive, but fail to terminate. Consider the following program:

```
function diverge()
{
  val r := ref(id)
  function foo() { (!r)() }
  r := foo
  foo()
}
```

In this function, we first create a reference r initialized with the identify function. Next we define a local function foo which calls the function in r. Then we assign foo itself to r and call foo, which will now never terminate even though there is no syntactic recursion.

But how can we infer in general that $diverge$ must include the div effect? It turns out that in essence reading from the heap may result in divergence. A conservative approach would be to assign the div effect to the type of read (!). For simplicity, this is what we will do in the formal development.

But in Koka we use a more sophisticated approach. In order to cause divergence, we actually need a higher-order heap where we read a function from the heap which accesses the heap itself. Fortunately, our effect system makes this behavior already apparent in the inferred types! – in our example, the effect of foo contains $read\langle h\rangle$, which is being stored in a reference in the same heap of type $ref\langle h, () \rightarrow read\langle h\rangle\ ()\rangle$. Note how the heap parameter h is itself present in the type of the values that this reference stores.

The trick is now that we generate a type constraint $hdiv\langle h, \tau, \epsilon\rangle$ for every heap read that keeps track of heap type h, the type of the value that was read τ and the current effect ϵ. The constraint $hdiv\langle h, \tau, \epsilon\rangle$ expresses that if $h \in \mathsf{ftv}(\tau)$ then the effect ϵ must include divergence. In particular, this constraint is fine-grained enough that any reading of a non-function type, or non-stateful functions will never cause divergence (and we can dismiss the constraint) The drawback is that if τ is polymorphic at generalization time, we need to keep the constraint around (as we cannot decide at that point whether h will ever be in $\mathsf{ftv}(\tau)$), which in turn means we need to use a system of qualified types [11]. Currently this is not fully implemented yet in Koka, and if at generalization time we cannot guarantee τ will never contain a reference to the heap h, we conservatively assume that the function may diverge.

2.8. Koka in practice

When designing a new type system it is always a question how well it will work in practice: does it infer the types you expect? Do the types become too complicated? Is the termination checker strong enough? etc. We have implemented the effect inference and various extensions in the Koka language which is freely available on the web. The Koka system currently has a JavaScript backend and can generate code that runs on NodeJS or inside a web page. We have written many small to medium sized samples to see how well the system works in practice.

2.8.1. Markdown

One application is a fully compliant Markdown text processor. This program consists of three phases where it first parser block elements, performs block analysis, collecting link definitions, numbering sections etc, and finally renders the inline elements in each block. The program passes the full Markdown test suite. Remarkably, almost all functions are inferred to be $total$, and only a handful of driver functions perform side effects, like reading input files. For efficiency though, many internal functions use local state. For example, when rendering all inline elements in a block, we use a local mutable string builder (of type $builder\langle h\rangle$) to build the result string in constant time (actual Koka code):

```
function formatInline( ctx : inlineCtx, txt : string) : string {
  formatAcc(ctx, builder(), txt)
}

function formatAcc( ctx : inlineCtx, acc : builder⟨h⟩,
                    txt : string ) : st⟨h⟩ string
{
  if (txt == "") return acc.string
  val (s, next) = matchRules(ctx.grammar, ctx, txt, id)
  formatAcc(ctx, acc.append(s), txt.substr1(next))
}
```

Note how $formatAcc$ is stateful due to the calls to the $append$ and $string$ methods of the string builder acc, but the outer function $formatInline$ is inferred to be $total$ since Koka can automatically apply the run function and encapsulate the state: indeed it is not observable if we use a mutable string builder internally or not. This pattern also occurs for example in the block analysis phase where we use a mutable hashmap to build the dictionary of link definitions.

As an aside, it turns out that on NodeJS, our Markdown program is about 30% faster than the fastest JavaScript Markdown processor at this time (*marked*), and about 6 times faster than the most widely used one (*showdown*). Using pure functional programs seems not only good for programmers, but also for JavaScript interpreters!

2.8.2. Safe tier-splitting

Most of the HTML5 DOM and the Node API's are available in Koka which allows us to write more substantial programs and evaluate the effect inference system in practice. We use two new effects for most external functions: the effect dom for functions that may have any side effect through a DOM call, and the effect $blocking$ for calls in NodeJS that may block (like $readFileSync$ for example).

On the web, many programs are split in a server and client part communicating with each other using some data encoding like JSON. It is advantageous to write both the client and server part as one program. In particular, the client and server part can share one common type definition for the data they exchange ensuring that they are always in sync and enabling automatic decoding and encoding of that data (depending on language support). Also, their interaction will be more apparent they can share common functionality, like date parsing, ensuring that both parts behave similarly.

Safely splitting a program into a server and client part is difficult though. For example, the client code may call a library function that itself calls a function that can only be run on the server (like writing to a log file), or the other way around. Moreover, if the client and server part both access a shared global variable (or both call a library function that uses an internal global variable) then we cannot split this code anymore.

The Koka effect types tackle both problems though and enable safe tier splitting. In particular, our main tier splitting function has the following (simplified) type signature:

```
function tiersplit(
  serverPart : () → server (
                      (α→ server ()) → server (β→ server ()) ),
  clientPart : (β→ client ()) → client (α→ client ())
) : io ()
```

where the $server$ and $client$ effects are defined as:

```
alias server = io
alias client = ⟨dom, div⟩
```

The $tiersplit$ function takes a server and client function and sets up a socket connection. On the server it will call the server part function which can initialize. Now, both the client and server part can be called for each fresh connection where $tiersplit$ supplies a $send$ function that takes a message of type α for client messages, and β for server messages. Both the client and server part return a fresh 'connection' function that handles incoming messages from the server or client respectively. Note how this type guarantees that messages sent to the client, and messages handled by the client, are both of type α, while for the server messages they will be β.

Furthermore, because the effect types for $server$ and $client$ are closed, the client and server part will only be able to call functions available for the client or server respectively. For example, if the client tries to call $print$ it will get the $console$ effect which will not unify with the $client$ effect, statically rejecting the program. Similarly for the server part if it tries to call functions with the dom effect (like $alert$). However, any function with a $total$ or $pure$ effect, like $length$ or map, can be called by both the server and client part and they can share common functionality.

$$
\begin{array}{llll}
\text{kinds} & \kappa & ::= & * & \text{value types} \\
 & & \mid & \mathsf{e} & \text{effect rows} \\
 & & \mid & \mathsf{k} & \text{effect constant} \\
 & & \mid & \mathsf{h} & \text{heap types} \\
 & & \mid & (\kappa_1, ..., \kappa_n) \rightarrow \kappa & \text{type constructor} \\
 & & & & \\
\text{types} & \tau^k & ::= & \alpha^k & \text{type variable} \\
 & & \mid & c^\kappa & \text{type constant} \\
 & & \mid & c^{\kappa_0}\langle \tau_1^{\kappa_1}, ..., \tau_n^{\kappa_n} \rangle & \kappa_0 = (\kappa_1,...,\kappa_n) \rightarrow \kappa \\
 & & & & \\
\text{schemes} & \sigma & ::= & \forall \alpha_1 ... \alpha_n .\, \tau^* &
\end{array}
$$

$$
\begin{array}{lllll}
\text{constants} & () & :: & * & \text{unit type} \\
 & (\_\!\rightarrow \_\,\_) & :: & (*, \mathsf{e}, *) \rightarrow * & \text{functions} \\
 & \langle\rangle & :: & \mathsf{e} & \text{empty effect} \\
 & \langle\_|\_\rangle & :: & (\mathsf{k}, \mathsf{e}) \rightarrow \mathsf{e} & \text{effect extension} \\
 & ref & :: & (\mathsf{h}, *) \rightarrow * & \text{references} \\
 & exn & :: & \mathsf{k} & \text{partial} \\
 & div & :: & \mathsf{k} & \text{divergent} \\
 & st & :: & \mathsf{h} \rightarrow \mathsf{k} & \text{stateful} \\
 & \ldots & & & \\
 & & & & \\
\text{notation} & \mu & & \text{effect variable} & \alpha^{\mathsf{e}} \\
 & \epsilon & & \text{effect type} & \tau^{\mathsf{e}} \\
 & l & & \text{effect constant type} & \tau^{\mathsf{e}} \\
 & \xi & & \text{heap variable} & \alpha^{\mathsf{h}} \\
 & \text{h} & & \text{heap type} & \tau^{\mathsf{h}}
\end{array}
$$

$$
\begin{array}{llcl}
\text{syntactic sugar} & \tau_1 \rightarrow \tau_2 & = & \tau_1 \rightarrow \langle\rangle\ \tau_2 \\
 & \langle l_1, ..., l_n \mid \epsilon \rangle & = & \langle l_1 \ldots \langle l_n \mid \epsilon \rangle \ldots \rangle \\
 & \langle l_1, ..., l_n \rangle & = & \langle l_1, ..., l_n \mid \langle\rangle \rangle
\end{array}
$$

Figure 1. Syntax of types and kinds. An extra restriction is that effect constants cannot be type variables, i.e. α^{k} is illegal.

Finally, the Koka effect system also prevents accidental sharing of global state by the client and server part. Both the client and server can use state that is contained in their handler. In that case the $st\langle h\rangle$ effect will be inferred, and discarded because h will generalize. However, if either function tries to access a shared variable in an outer scope, then the h will *not* generalize (because the variable will have type $ref\langle h, a\rangle$ and therefore h is not free in the environment), in which case the inferred $st\langle h\rangle$ effect cannot be removed. Again, this will lead to a unification failure and the program will be statically rejected.[2]

3. The type system

In this section we are going to give a formal definition of our polymorphic effect system for a small core-calculus that captures the essence of Koka. We call this λ^k. Figure 1 defines the syntax of types. The well-formedness of types τ is guaranteed by a simple kind system. We put the kind κ of a type τ in superscript, as τ^κ. We have the usual kind $*$ and $\rightarrow$, but also kinds for effect rows

[2] Actually, in the real implementation, both io and dom include the st effect but each with a different heap constant, namely $ioheap$ and $hdom$ respectively; still causing a unification error at some point since these will not unify with each other.

$$
\begin{array}{ll}
(\textsc{eq-refl}) & \epsilon \equiv \epsilon \\[1em]
(\textsc{eq-trans}) & \dfrac{\epsilon_1 \equiv \epsilon_2 \quad \epsilon_2 \equiv \epsilon_3}{\epsilon_1 \equiv \epsilon_3} \\[1em]
(\textsc{eq-head}) & \dfrac{l_1 \equiv l_2 \quad \epsilon_1 \equiv \epsilon_2}{\langle l_1 \mid \epsilon_1 \rangle \equiv \langle l_2 \mid \epsilon_2 \rangle} \\[1em]
(\textsc{eq-swap}) & \dfrac{l_1 \not\equiv l_2}{\langle l_1 \mid \langle l_2 \mid \epsilon \rangle\rangle \equiv \langle l_2 \mid \langle l_1 \mid \epsilon \rangle\rangle} \\[1em]
(\textsc{eq-lab}) & c\langle \tau_1, ..., \tau_n \rangle \equiv c\langle \tau_1', ..., \tau_n' \rangle
\end{array}
$$

Figure 2. Effect equivalence.

(e), effect constants (k), and heaps (h). Often the kind of a type is immediately apparent or not relevant, and most of the time we will not denote the kind to reduce clutter, and just write plain types τ. For clarity, we are using α for regular type variables, μ for effect type variables, and ξ for heap type variables.

Effect types are defined as a row of effect labels l. Such effect row is either empty $\langle\rangle$, a polymorphic effect variable μ, or an extension of an effect row ϵ with an effect constant l, written as $\langle l|\epsilon\rangle$. The effect constants can be anything that is interesting to our language. For our purposes we will restrict the constants to exceptions (exn), divergence (div), and heap operations (st). It is no problem to generalize this to the more fine-grained hierarchy of Koka but this simplifies the presentation and proofs. The kind system ensures that an effect is always either a *closed effect* of the form $\langle l_1, ..., l_n\rangle$, or an *open effect* of the form $\langle l_1, ..., l_n \mid \mu\rangle$.

Figure 2 defines an equality relation $\quad =$ between effect types. In particular, the equations encode that we consider effects equivalent regardless of the order of the effect constants. In contrast to many record calculi, for example [6, 28, 25], effect rows *do* allow duplicate labels where an effect $\langle exn, exn\rangle$ is allowed (and not equal to the effect $\langle exn\rangle$). The definition of effect equality is essentially the same as for scoped labels [14] where we ignore the type components. Note that (eq-lab) defines equality over the effect constants where the type arguments are not taken into account. Most constants have no arguments and thus compare directly (as $c^{\mathsf{k}} \equiv c^{\mathsf{k}}$). The only exception in our system is the state effect where $st\langle h_1\rangle \equiv st\langle h_2\rangle$ for any h_1 and h_2.

Using effect equality, we define the notation $l \in \epsilon$ as:

$$
l \in \epsilon \quad \text{iff} \quad \epsilon \equiv \langle l \mid \epsilon' \rangle \quad \text{for some } \epsilon'
$$

3.1. Type rules

Figure 3 defines the formal type rules of our effect system. The rules are defined over a small expression calculus:

$$
\begin{array}{lcll}
e & ::= & x & \text{(variables)} \\
 & | & p & \text{(primitives)} \\
 & | & e_1\, e_2 & \text{(application)} \\
 & | & \lambda x.\ e & \text{(function)} \\
 & | & x \leftarrow e_1;\ e_2 & \text{(sequence)} \\
 & | & \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 & \text{(let binding)} \\
 & | & \mathsf{catch}\ e_1\ e_2 & \text{(catch exceptions)} \\
 & | & \mathsf{run}\ e & \text{(isolate)}
\end{array}
$$

$$
p \ ::= \ () \mid \mathsf{fix} \mid \mathsf{throw} \mid \mathsf{new} \mid (!) \mid (:=)
$$

$$
\begin{array}{ll}
(\textsc{var}) & \dfrac{\Gamma(x)\ =\ \sigma}{\Gamma \vdash\ x : \sigma \mid \epsilon} \\[1em]
(\textsc{lam}) & \dfrac{\Gamma, x : \tau_1 \vdash\ e : \tau_2 \mid \epsilon_2}{\Gamma \vdash\ \lambda x.\, e : \tau_1\ \to\ \epsilon_2\ \tau_2 \mid \epsilon} \\[1em]
(\textsc{app}) & \dfrac{\Gamma \vdash\ e_1 : \tau_2\ \to\ \epsilon\ \tau \mid \epsilon \quad \Gamma \vdash\ e_2 : \tau_2 \mid \epsilon}{\Gamma \vdash\ e_1\, e_2 : \tau \mid \epsilon} \\[1em]
(\textsc{let}) & \dfrac{\Gamma \vdash\ e_1 : \sigma \mid \langle\rangle \quad \Gamma, x : \sigma \vdash\ e_2 : \tau \mid \epsilon}{\Gamma \vdash\ \mathit{let}\ x\ =\ e_1\ \mathit{in}\ e_2 : \tau \mid \epsilon} \\[1em]
(\textsc{gen}) & \dfrac{\Gamma \vdash\ e : \tau \mid \langle\rangle \quad \overline{\alpha} \notin \mathsf{ftv}(\Gamma)}{\Gamma \vdash\ e : \forall\overline{\alpha}.\, \tau \mid \langle\rangle} \\[1em]
(\textsc{inst}) & \dfrac{\Gamma \vdash\ e : \forall\overline{\alpha}.\, \tau \mid \epsilon}{\Gamma \vdash\ e : [\overline{\alpha} := \overline{\tau}]\tau \mid \epsilon} \\[1em]
(\textsc{run}) & \dfrac{\Gamma \vdash\ e : \tau \mid \langle st\langle\xi\rangle \mid \epsilon\rangle \quad \xi \notin \mathsf{ftv}(\Gamma, \tau, \epsilon)}{\Gamma \vdash\ \mathsf{run}\, e : \tau \mid \epsilon} \\[1em]
(\textsc{catch}) & \dfrac{\Gamma \vdash\ e_1 : \tau \mid \langle exn \mid \epsilon\rangle \quad \Gamma \vdash\ e_2 : () \to \epsilon\, \tau \mid \epsilon}{\Gamma \vdash\ \mathsf{catch}\, e_1\ e_2 : \tau \mid \epsilon}
\end{array}
$$

$$
\begin{array}{llll}
(\textsc{alloc}) & \Gamma\ \vdash\ \mathsf{ref} & : \tau\ \to\ \langle st\langle h\rangle \mid \epsilon\rangle\ \mathit{ref}\langle h, \tau\rangle \mid \epsilon' \\
(\textsc{read}) & \Gamma\ \vdash\ (!) & : \mathit{ref}\langle h, \tau\rangle\ \to\ \langle st\langle h\rangle, div \mid \epsilon\rangle\ \tau \mid \epsilon' \\
(\textsc{write}) & \Gamma\ \vdash\ (:=) & : (\mathit{ref}\langle h, \tau\rangle, \tau)\ \to\ \langle st\langle h\rangle \mid \epsilon\rangle\ () \mid \epsilon' \\
(\textsc{throw}) & \Gamma\ \vdash\ \mathsf{throw} & : ()\ \to\ \langle exn \mid \epsilon\rangle\ \tau \mid \epsilon' \\
(\textsc{unit}) & \Gamma\ \vdash\ () & : () \mid \epsilon \\
(\textsc{fix}) & \Gamma\ \vdash\ \mathit{fix} & : ((\tau_1 \to \langle div \mid \epsilon\rangle\, \tau_2) \\
 & & \qquad \to (\tau_1 \to \langle div \mid \epsilon\rangle\, \tau_2)) \\
 & & \qquad\quad \to (\tau_1 \to \langle div \mid \epsilon\rangle\, \tau_2) \mid \epsilon'
\end{array}
$$

Figure 3. General type rules with effects.

This expression syntax is meant as a surface syntax, but when we discuss the semantics of the calculus, we will refine and extend the syntax further (see Figure 7). We use the bind (or sequence) expression $x \leftarrow e_1;\ e_2$ for a monomorphic binding of a variable x to an expression e_1. This is just syntactic sugar for the application $(\lambda x.\ e_2)\ e_1$. We write $e_1; e_2$ as a shorthand for the expression $x \leftarrow e_1;\ e_2$ where $x \notin \mathsf{fv}(e_2)$. We have added run and catch as special expressions since this simplifies the presentation as where we can give direct type rules for them. Also, we simplified both catch and throw by limiting the \textit{exception} type to just the unit type (()).

The type rules are stated under a type environment Γ which maps variables to types. An environment can be extended using a comma. If Γ' is equal to $\Gamma, x : \sigma$ then $\Gamma'(x) = \sigma$ and $\Gamma'(y) = \Gamma(y)$ for any $y \neq x$. A type rule of the form $\Gamma\ \vdash\ e\ :\ \sigma\ \mid\ \epsilon$ states that under an environment Γ the expression e has type σ with an effect ϵ.

Most of the type rules in Figure 3 are quite standard. The rule (var) derives the type of a variable. The derived effect is any arbitrary effect ϵ. We may have expected to derive only the total effect $\langle\rangle$ since the evaluation of a variable has no effect at all (in our strict setting). However, there is no rule that lets us upgrade the final effect and instead we get to pick the final effect right away. Another way to look at this is that since the variable evaluation has no effect, we are free to assume any arbitrary effect.

The (lam) rule is similar: the evaluation of a lambda expression is a value and has no effect and we can assume any arbitrary effect ϵ. Interestingly, the effect derived for the body of the lambda expression, ϵ_2, shifts from the derivation on to the derived function type $\tau_1\ \to\ \epsilon_2\ \tau_2$: indeed, calling this function and evaluating the body causes the effect ϵ_2. The (app) is also standard, and derives an effect ϵ requiring that its premises derive the same effect as the function effect.

Instantiation ((inst)) is standard and instantiates a type scheme. The generalization rule (gen) has an interesting twist: it requires

$$\text{(VAR)}_{\mathrm{s}} \qquad \frac{\Gamma(x) \;=\; \forall\overline{\alpha}.\,\tau}{\Gamma \vdash_s \; x : [\overline{\alpha} \mapsto \overline{\tau}]\tau \mid \epsilon}$$

$$\text{(LET)}_{\mathrm{s}} \qquad \frac{\begin{array}{c}\Gamma \vdash_s \; e_1 : \tau_1 \mid \langle\rangle \quad \overline{\alpha} \notin \mathsf{ftv}(\Gamma) \\ \Gamma, x : \forall\overline{\alpha}.\,\tau_1 \vdash_s \; e_2 : \tau_2 \mid \epsilon\end{array}}{\Gamma \vdash_s \; \mathsf{let}\; x \;=\; e_1 \;\mathsf{in}\; e_2 : \tau_2 \mid \epsilon}$$

Figure 4. Changed rules for the syntax directed system; Rule (INST) and (GEN) are removed, and all other rules are equivalent to the declarative system (Figure 3)

the derived effect to be total $\langle\rangle$. It turns out this is required to ensure a sound semantics as we show in Section 5. Indeed, this is essentially the equivalent of the value restriction in ML [16]. Of course, in ML effects are not inferred by the type system and the value restriction syntactically restricts the expression over which one can generalize. In our setting we can directly express that we only generalize over total expressions. As we see in Section 6.1, we can give a direct semantic interpretation of why this restriction is necessary since without it, we cannot prove subject reduction. The rule (LET) binds expressions with a polymorphic type scheme σ and just like (GEN) requires that the bound expression has no effect. It turns out that is still sound to allow more effects at generalization and let bindings. In particular, we can allow exn, div. However, for the formal development we will only consider the empty effect for now.

All other rules are just type rules for the primitive constants. Note that all the effects for the primitive constants are open and can be freely chosen (just as in the (VAR) rule). This is important as it allows us to always assume more effects than induced by the operation.

4. Type inference

As a first step toward type inference, we first present in a syntax directed version of our declarative type rules in Figure 4. For this system, the syntax tree completely determines the derivation tree. Effectively, we removed the (INST) and (GEN) rules, and always apply instantiation in the (VAR) rule, and always generalize at let-bindings. This technique is entirely standard [10, 20, 11] and we can show that the syntax directed system is sound and complete with respect to the declarative rules:

Theorem 1. (*Soundness of the syntax directed rules*)
When $\Gamma \vdash_s \; e : \tau \mid \epsilon$ then we also have $\Gamma \vdash \; e : \tau \mid \epsilon$.

Theorem 2. (*Completeness of the syntax directed rules*)
When $\Gamma \vdash \; e : \sigma \mid \epsilon$ then we also have $\Gamma \vdash \; e : \tau \mid \epsilon$ where σ can be instantiated to τ.

Both proofs are by straightforward induction using standard techniques as described for example by Jones [11].

4.1. The type inference algorithm

Starting from the syntax directed rules, we can now give a the type inference algorithm for our effect system which is shown in Figure 5. Following Jones [11] we present the algorithm as natural inference rules of the form $\theta\Gamma \vdash \; e : \tau \mid \epsilon$ where θ is a substitution, Γ the environment, and e, τ, and ϵ, the expression, its type, and its effect respectively. The rules can be read as an attribute grammar where θ, τ, and ϵ are synthesised, and Γ and e inherited. An advantage is that this highlights the correspondence between the syntax directed rules and the inference algorithm.

The algorithm uses unification written as $\tau_1 \sim \tau_2 \;\; : \theta$ which unifies τ_1 and τ_2 with a most general substitution θ such that

$$\text{(VAR)}_{\mathrm{i}} \qquad \frac{\Gamma(x) \;=\; \forall\overline{\alpha}.\,\tau}{\varnothing\Gamma \vdash_i \; x : [\overline{\alpha} \mapsto \overline{\beta}]\tau \mid \mu}$$

$$\text{(LAM)}_{\mathrm{i}} \qquad \frac{\theta\Gamma, x : \alpha \vdash_i \; e : \tau_2 \mid \epsilon_2}{\theta\Gamma \vdash_i \; \lambda x.\, e : \theta\alpha \;\to\; \epsilon_2 \; \tau_2 \mid \mu}$$

$$\text{(APP)}_{\mathrm{i}} \qquad \frac{\begin{array}{c}\theta_1\Gamma \vdash_i \; e_1 : \tau_1 \mid \epsilon_1 \quad \theta_2(\theta_1\Gamma) \vdash_i \; e_2 : \tau_2 \mid \epsilon_2 \\ \theta_2\, \tau_1 \sim (\tau_2 \;\to\; \epsilon_2 \; \alpha) \;\; : \theta_3 \quad \theta_3\theta_2\epsilon_1 \sim \theta_3\epsilon_2 \;\; : \theta_4\end{array}}{\theta_4\theta_3\theta_2\theta_1\Gamma \vdash_i \; e_1 \, e_2 : \theta_4\theta_3\alpha \mid \theta_4\theta_3\epsilon_2}$$

$$\text{(LET)}_{\mathrm{i}} \qquad \frac{\begin{array}{c}\theta_1\Gamma \vdash_i \; e_1 : \tau_1 \mid \epsilon_1 \quad \epsilon_1 \sim \langle\rangle \;\; : \theta_2 \\ \sigma = \; \mathsf{gen}(\theta_2\theta_1\Gamma, \theta_2\tau_1) \\ \theta_3(\theta_2\theta_1\Gamma, x : \sigma) \vdash_i \; e_2 : \tau \mid \epsilon\end{array}}{\theta_3\theta_2\theta_1\Gamma \vdash_i \; \mathit{let}\; x \;=\; e_1 \;\mathit{in}\; e_2 : \tau \mid \epsilon}$$

$$\text{(RUN)}_{\mathrm{i}} \qquad \frac{\begin{array}{c}\theta_1\Gamma \vdash_i \; e : \tau \mid \epsilon \quad \epsilon \sim \langle st\langle\xi\rangle \mid \mu\rangle \;\; : \theta_2 \\ \theta_2\xi \in \mathit{TypeVar} \quad \theta_2\xi \notin \mathsf{ftv}(\theta_2\theta_1\Gamma, \theta_2\tau, \theta_2\mu)\end{array}}{\theta_2\theta_1\Gamma \vdash_i \; \mathsf{run}\, e : \theta_2\tau \mid \theta_2\mu}$$

$$\text{(CATCH)}_{\mathrm{i}} \qquad \frac{\begin{array}{c}\theta_1\Gamma \vdash_i \; e_1 : \tau_1 \mid \epsilon_1 \quad \theta_2(\theta_1\Gamma) \vdash_i \; e_2 : \tau_2 \mid \epsilon_2 \\ \theta_2\epsilon_1 \sim \langle exn \mid \epsilon_2\rangle \;\; : \theta_3 \\ \theta_3\tau_2 \sim () \to \theta_3\epsilon_2 \; \theta_3\theta_2\tau_1 \;\; : \theta_4\end{array}}{\theta_4\theta_3\theta_2\theta_1\Gamma \vdash \; \mathsf{catch}\, e_1 \; e_2 : \theta_4\theta_3\tau_2 \mid \theta_4\theta_3\epsilon_2}$$

Figure 5. Type and effect inference algorithm. Any type variables α, μ, ξ, and $\overline{\alpha}$ are considered fresh.

$$\text{(UNI-VAR)} \qquad \alpha \sim \alpha \;\; : []$$

$$\text{(UNI-VARL)} \qquad \frac{\alpha \notin \mathsf{ftv}(\tau)}{\alpha^k \sim \tau^k \;\; : [\alpha \mapsto \tau]}$$

$$\text{(UNI-VARR)} \qquad \frac{\alpha \notin \mathsf{ftv}(\tau)}{\tau^k \sim \alpha^k \;\; : [\alpha \mapsto \tau]}$$

$$\text{(UNI-CON)} \qquad \frac{\begin{array}{c}\forall i \; \in 1..n. \quad \theta_{i-1}...\theta_1\tau_i \sim \theta_{i-1}...\theta_1 t_i \;\; : \theta_i \\ \kappa = \; (\kappa_1, ..., \kappa_n) \to \kappa'\end{array}}{c^\kappa\langle\tau_1^{\kappa_1}, ..., \tau_n^{\kappa_n}\rangle \sim c^\kappa\langle t_1^{\kappa_1}, ..., t_n^{\kappa_n}\rangle \;\; : \theta_n...\theta_1}$$

$$\text{(UNI-EFF)} \qquad \frac{\begin{array}{c}\epsilon_2 \simeq l \mid \epsilon_3 : \theta_1 \quad \mathsf{tl}(\epsilon_1) \; \notin \mathsf{dom}(\theta_1) \\ \theta_1\epsilon_1 \sim \theta_1\epsilon_3 \;\; : \theta_2\end{array}}{\langle l \mid \epsilon_1\rangle \sim \epsilon_2 \;\; : \theta_2\theta_1}$$

$$\text{(EFF-HEAD)} \qquad \frac{l \;\equiv l' \quad l \sim l' \;\; : \theta}{\langle l' \mid \epsilon\rangle \simeq l \mid \epsilon : \theta}$$

$$\text{(EFF-SWAP)} \qquad \frac{l \;\not\equiv l' \quad \epsilon \simeq l \mid \epsilon' : \theta}{\langle l' \mid \epsilon\rangle \simeq l \mid \langle l \mid \epsilon'\rangle : \theta}$$

$$\text{(EFF-TAIL)} \qquad \frac{\text{fresh } \mu'}{\mu \simeq l \mid \mu' : [\mu \mapsto \langle l \mid \mu'\rangle]}$$

Figure 6. Unification: $\tau \sim \tau' \;\; : \theta$ unifies two types and returns a substitution θ. It uses effect unification $\epsilon \simeq l \mid \epsilon' : \theta$ which takes an effect ϵ and effect primitive l as input, and returns effect tail ϵ' and a substition θ.

$\theta\tau_1 = \theta\tau_2$. %The unification algorithm is standard and effects are unified using standard row unificiation allowing for duplicate label as described by Leijen [14]. The gen function generalizes a type with respect to an environment and is defined as:

$$\mathsf{gen}(\Gamma, \tau) = \forall(\mathsf{ftv}(\tau) - \mathsf{ftv}(\Gamma)).\, \tau$$

We can prove that the inference algorithm is sound and complete with respect to the syntax directed rules (and by Theorem 1 and 2 also sound and complete to the declarative rules):

Theorem 3. (*Soundness*)
If $\theta\Gamma \vdash_i\ e : \tau \mid \epsilon$ then there exists a θ' such that $\theta\Gamma \vdash_s\ e : \tau' \mid \epsilon'$ where $\theta'\tau = \tau'$ and $\theta'\epsilon = \epsilon'$.

Theorem 4. (*Completeness*)
If $\theta_1\Gamma \vdash_s\ e : \tau_1 \mid \epsilon_1$ then $\theta_2\Gamma \vdash_i\ e : \tau_2 \mid \epsilon_2$ and there exists a substitution θ such that $\theta_1 \approx \theta\theta_2$, $\tau_1 = \theta\tau_2$ and $\epsilon_1 = \theta\epsilon_2$.

Since the inference algorithm is basically just algorithm W [4] together with extra unifications for effect types, the proofs of soundness and completeness are entirely standard. The main extended lemma is for the soundness, completeness, and termination of the unification algorithm which now also unifies effect types.

The unification algorithm is shown in Figure 6. The algorithm is an almost literal adaption of the unification algorithm for records with scoped labels as described by Leijen [14], and the proofs of soundness, completeness, and termination carry over directly.

The first four rules are the standard Robinson unification rules with a slight modification to return only kind-preserving substitutions [6, 12]. The rule (UNI-EFF) unifies effect rows. The operation $\mathsf{tl}(\epsilon)$ is defined as:

$$\begin{array}{lcl}
\mathsf{tl}(\langle l_1, ..., l_n \mid \mu\rangle) & = & \mu\\
\mathsf{tl}(\langle l_1, ..., l_n\rangle) & = & \langle\rangle
\end{array}$$

As described in detail in [14], the check $\mathsf{tl}(\epsilon_1) \notin \mathsf{dom}(\theta)_1$ is subtle but necessary to guarantee termination of row unification. The final three rules unify an effect with a specific head. In particular, $\epsilon \simeq l \mid \epsilon' : \theta$ states that for a given effect row ϵ, we match it with a given effect constant l, and return an effect tail ϵ' and substitution θ such that $\theta\epsilon = \langle\theta l \mid \theta\epsilon'\rangle$. Each rule basically corresponds to the equivalence rules on effect rows (Figure 2).

5. Semantics of effects

In this section we are going to define a precise semantics for our language, and show that well-typed programs cannot go 'wrong'. In contrast to our earlier soundness and completeness result for the type inference algorithm, the soundness proof of the type system in Hindley-Milner does not carry over easily in our setting: indeed, we are going to model many complex effects which is fraught with danger.

First, we strengthen our expression syntax by separating out value expressions v, as shown in Figure 7. We also define basic values b as values that cannot contain expressions themselves. Moreover, we added a few new expressions, namely heap bindings ($\mathsf{hp}\,\varphi.\,e$), a partially applied catch ($\mathsf{catch}\,e$), a partially applied assignments (v :=), and general constants (c). Also, we denote heap variables using r. An expression $\mathsf{hp}\,\langle r_1\ \mapsto\ v_1\rangle, ..., \langle r_n\ \mapsto\ v_n\rangle.\ e$ binds r_1 to r_n in $v_1, ..., v_n$ and e. By convention, we always require r_1 to r_n to be distinct, and consider heaps φ equal modulo alpha-renaming.

$$\begin{array}{lcll}
e & ::= & v & \text{(value)}\\
 & | & e_1\ e_2 & \text{(application)}\\
 & | & \mathsf{let}\ x\ =\ e_1\ \mathsf{in}\ e_2 & \text{(let binding)}\\
 & | & \mathsf{hp}\,\varphi.\,e & \text{(heap binding)}\\
 & | & \mathsf{run}\,e & \text{(isolate)}\\
 & & & \\
v & ::= & \lambda x.\,e & \text{(function)}\\
 & | & \mathsf{catch}\,e & \text{(partial catch)}\\
 & | & b & \text{(basic value (contains no } e\text{))}\\
 & & & \\
b & ::= & x & \text{(variable)}\\
 & | & c & \text{(constant)}\\
 & | & \mathsf{fix} & \text{(fixpoint)}\\
 & | & \mathsf{throw} & \text{(throw an exception)}\\
 & | & \mathsf{catch} & \text{(catch exceptions)}\\
 & | & r & \text{(reference variable)}\\
 & | & \mathsf{ref} & \text{(new reference)}\\
 & | & (!) & \text{(dereference)}\\
 & | & (:=) & \text{(assign)}\\
 & | & (r\ :=) & \text{(partial assign)}\\
 & & & \\
w & ::= & b \mid \mathsf{throw}\ v & \text{(basic value or exception)}\\
a & ::= & v \mid \mathit{throw}\ v \mid \mathsf{hp}\,\varphi.\ v \mid \mathsf{hp}\,\varphi.\,\mathsf{throw}\ v & \text{(answers)}\\
 & & & \\
\varphi & ::= & \langle r_1\ \mapsto v_1\rangle\ ...\ \langle r_n\ \mapsto v_n\rangle & \text{(heap bindings)}
\end{array}$$

Figure 7. Full expression syntax

$$\text{(HEAP)}\quad \frac{\begin{array}{c}\forall\langle r_i\ \mapsto v_i\rangle\ \in \varphi.\ \ \Gamma, \overline{\varphi}_h \vdash\ v_i : \tau_i \mid \langle\rangle\\ \Gamma, \overline{\varphi}_h \vdash\ e : \tau \mid \langle st\langle h\rangle \mid \epsilon\rangle\end{array}}{\Gamma \vdash\ \mathsf{hp}\,\varphi.\, e : \tau \mid \langle st\langle h\rangle \mid \epsilon\rangle}$$

$$\text{(EXTEND)}\quad \frac{\Gamma \vdash\ e : \tau \mid \epsilon}{\Gamma \vdash\ e : \tau \mid \langle l \mid \epsilon\rangle}$$

$$\text{(CONST)}\quad \frac{\mathsf{typeof}(c)\ =\ \sigma}{\Gamma \vdash\ c : \sigma \mid \epsilon}$$

Figure 8. Extra type rules for heap expressions and constants. We write $\overline{\varphi}_h$ for the conversion of a heap φ to a type environment: if φ equals $\langle r_1\ \mapsto v_1, ..., r_n\ \mapsto v_n\rangle$ then $\overline{\varphi}_h = r_1 : \mathit{ref}\langle h, \tau_1\rangle, ..., r_n : \mathit{ref}\langle h, \tau_n\rangle$ for some τ_1 to τ_n.

The surface language will never expose the heap binding construct $\mathsf{hp}\,\varphi.\,e$ directly to the user but during evaluation the reductions on heap operations will create heaps and use them. In order to give a type to such expression, we need an extra type rule for heap bindings, given in Figure 8. Note how each heap value is typed under an enviroment that contains types for all bindings (much like a recursive let binding). Moreover, a heap binding induces the stateful effect $st\langle h\rangle$. The (EXTEND) rule states that we can always assume a worse effect; this rule is not part of the inference rules but we need it to show subject reduction of stateful computations. The same figure also defines the type rule for constants where we assume a function $\mathsf{typeof}(c)$ that returns a closed type scheme for each constant.

Finally, we note that for all value expression, we can assume any effect type, including the empty effect:

Lemma 1. (*Value typing*)
If a value is well-typed, $\Gamma \vdash\ v : \tau \mid \epsilon$ then also $\Gamma \vdash\ v : \tau \mid \langle\rangle$.

5.1. Reductions

We can now consider primitive reductions for the various expressions as shown in Figure 9. The first four reductions are standard for the lambda calculus. To abstract away from a particular set of constants, we assume a function δ which takes a constant and a closed value to a closed value. Following [38] we assume δ-typability for each constant: If $\mathsf{typeof}(c) = \forall\overline{\alpha}.\,\tau_1 \rightarrow \epsilon\tau_2$, with $\theta = [\overline{\alpha} \mapsto \overline{\tau}]$ and

$$\begin{array}{lllll}
(\delta) & c\;v & \longrightarrow & \delta(c,v) & \text{if } \delta(c,v) \text{ is def.}\\
(\beta) & (\lambda x.\,e)\;v & \longrightarrow & [x \mapsto v]e & \\
(\text{\textsc{let}}) & \mathsf{let}\;x \;=\; v\;\mathsf{in}\;e & \longrightarrow & [x\mapsto v]e & \\
 & & & & \\
(\text{\textsc{fix}}) & \mathsf{fix}\;v & \longrightarrow & v\;(\lambda x.\,(\mathsf{fix}\;v)\;x) & \\
 & & & & \\
(\text{\textsc{throw}}) & X[\mathsf{throw}\;v] & \longrightarrow & \mathsf{throw}\;v & \text{if } X \neq [\,]\\
(\text{\textsc{catcht}}) & \mathsf{catch}\;(\mathsf{throw}\;v)\;e & \longrightarrow & e\;v & \\
(\text{\textsc{catchv}}) & \mathsf{catch}\;v\;e & \longrightarrow & v & \\
 & & & & \\
(\text{\textsc{alloc}}) & \mathsf{ref}\;v & \longrightarrow & \mathsf{hp}\,\langle r\mapsto v\rangle.\,r & \\
(\text{\textsc{read}}) & \mathsf{hp}\,\varphi\langle r\mapsto v\rangle.\,R[!r] & \longrightarrow & \mathsf{hp}\,\varphi\langle r\mapsto v\rangle.\,R[v] & \\
(\text{\textsc{write}}) & \mathsf{hp}\,\varphi\langle r\mapsto v_1\rangle.\,R[r := v_2] & \longrightarrow & \mathsf{hp}\,\varphi\langle r\mapsto v_2\rangle.\,R[()] & \\
 & & & & \\
(\text{\textsc{merge}}) & \mathsf{hp}\,\varphi_1.\,\mathsf{hp}\,\varphi_2.\,e & \longrightarrow & \mathsf{hp}\,\varphi_1\varphi_2.\,e & \\
(\text{\textsc{lift}}) & R[\mathsf{hp}\,\varphi.\,e] & \longrightarrow & \mathsf{hp}\,\varphi.\,R[e] & \text{if } R\neq[\,]\\
 & & & & \\
(\text{\textsc{runl}}) & \mathsf{run}\;[\mathsf{hp}\,\varphi.]\;\lambda x.\,e & \longrightarrow & \lambda x.\,\mathsf{run}\;([\mathsf{hp}\,\varphi.]\;e) & \\
(\text{\textsc{runc}}) & \mathsf{run}\;[\mathsf{hp}\,\varphi.]\;\mathsf{catch}\;e & \longrightarrow & \mathsf{catch}\;(\mathsf{run}\;([\mathsf{hp}\,\varphi.]\;e)) & \\
(\text{\textsc{runh}}) & \mathsf{run}\;[\mathsf{hp}\,\varphi.]\;w & \longrightarrow & w & \text{if } \mathsf{frv}(w) \not\pitchfork \mathsf{dom}(\varphi)
\end{array}$$

Evaluation contexts:

$$\begin{array}{lll}
X & ::= & [\,] \mid X\;e \mid v\;X \mid \mathsf{let}\;x \;=\; X\;\mathsf{in}\;e\\
R & ::= & [\,] \mid R\;e \mid v\;R \mid \mathsf{let}\;x \;=\; R\;\mathsf{in}\;e \mid \mathsf{catch}\;R\;e\\
E & ::= & [\,] \mid E\;e \mid v\;E \mid \mathsf{let}\;x \;=\; E\;\mathsf{in}\;e \mid \mathsf{catch}\;E\;e \mid \mathsf{hp}\,\varphi.\,E \mid \mathsf{run}\;E
\end{array}$$

Figure 9. Reduction rules and evaluation contexts.

$\cdot \vdash v : \theta\tau_1 \mid \langle\rangle$, then $\delta(c,v)$ is defined, and $\cdot \vdash \delta(c,v) : \theta\tau_2 \mid \theta\epsilon$. The reductions β and (LET) substitute the bound variable x with the evaluated value v in the body e. The (FIX) reduction is the fixpoint combinator and introduces recursion.

The next three rules deal with exceptions. In particular, the rule (THROW) progates exceptions under a context X. Since X does not include $\mathsf{catch}\;e_1\;e_2$, $\mathsf{hp}\,\varphi.\,e$ or $\mathsf{run}\;e$, this will propagate the exception to the nearest exception handler or state block. The (CATCHT) reduction catches exceptions and passes them on to the handler. If the handler raises an exception itself, that exception will then propagate to its nearest enclosing exception hander. In contrast to ML [38] we are not concerned with more complex exception types and assume here that exceptions are always of the unit type.

Following Wright and Felleisen [38] the next five rules model heap reductions. Allocation creates a heap, while (!) and (:=) read and write from the a heap. Through the R context, these always operate on the nearest enclosing heap since R does not contain $\mathsf{hp}\,\varphi.\,e$ or $\mathsf{run}\;e$ expressions. The rules (LIFT) and (MERGE) let us lift heaps out of expressions to ensure that all references can be bound in the nearest enclosing heap.

The final three rules deal with state isolation through run. We write $[\mathsf{hp}\,\varphi.]$ to denote an optional heap binding (so we really define six rules for state isolation). The first two rules (RUNL) and (RUNC) push a run operation down into a lambda-expression or partial catch expression.

The final rule (RUNH) captures the essence of state isolation and reduces to a new value (or exception) discarding the heap φ. The side condition $\mathsf{frv}(w) \not\pitchfork \mathsf{dom}(\varphi)$ is necessary to ensure well-formedness where a reference should not 'escape' its binding.

Using the reduction rules, we can now define an evaluation function. Using the evaluation context E defined in Figure 9, we define

$$E[e] \longmapsto E[e'] \quad \text{iff} \quad e \longrightarrow e'$$

The evaluation context ensures strict semantics where only the leftmost- outermost reduction is applicable in an expression. We define the relation $\mapsto\!\!\!\twoheadrightarrow$ as the reflexive and transtive closure of $\longmapsto$. We can show that $\mapsto\!\!\!\twoheadrightarrow$ is a function even though we need a simple diamond theorem since the order in which (LIFT) and (MERGE) reductions happen is not fixed [38].

The final results, or answers a, that expressions evaluate to, are either values v, exceptions $\mathsf{throw}\;v$, heap bound values $\mathsf{hp}\,\varphi.\,v$ or heap bound exceptions $\mathsf{hp}\,\varphi.\,\mathsf{throw}\;v$ (as defined in Figure 7).

6. Semantic soundness

We will now show that well-typed programs cannot go 'wrong'. Our proof closely follows the subject reduction proofs of Wright and Felleisen [38]. Most proofs are very similar except for the cases involving state isolation through run, and exception handling through catch where the exn effect can be discarded. Our main theorem is:

Theorem 5. (*Semantic soundness*)
If $\cdot \vdash e : \tau \mid \epsilon$ then either $e \Uparrow$ or $e \mapsto\!\!\!\twoheadrightarrow a$ where $\cdot \vdash a : \tau \mid \epsilon$.

where we use the notation $e \Uparrow$ for a never ending reduction. The proof of this theorem consists of showing two main lemmas:

- Show that reduction in the operational semantics preserves well-typing (called subject reduction).
- Show that *faulty* expressions are not typable.

If programs are closed and well-typed, we know from subject reduction that we can only reduce to well-typed terms, which can be either faulty, an answer, or an expression containing a further redex. Since faulty expressions are not typable it must be that evaluation either produces a well-typed answer or diverges. The above points are proven in the remainder of this section.

6.1. Subject reduction

The subject reduction theorem states that a well-typed term remains well-typed under reduction:

Lemma 2. (*Subject reduction*)
If $\Gamma \vdash e_1 : \tau \mid \epsilon$ and $e_1 \longrightarrow e_2$ then $\Gamma \vdash e_2 : \tau \mid \epsilon$.

To show that this holds, we need to establish various lemmas. Two particularly important lemmas are the substitution and extension lemmas:

Lemma 3. (*Substitution*)
If $\Gamma, x : \forall\overline{\alpha}.\,\tau \vdash e : \tau' \mid \epsilon$ where $x \notin \mathsf{dom}(\Gamma)$, $\Gamma \vdash v : \tau \mid \langle\rangle$, and $\overline{\alpha} \not\pitchfork \mathsf{ftv}(\Gamma)$, then $\Gamma \vdash [x \mapsto v]e : \tau' \mid \epsilon$.

Lemma 4. (*Extension*)
If $\Gamma \vdash e : \tau \mid \epsilon$ and for all $x \in \mathsf{fv}(e)$ we have $\Gamma(x) = \Gamma'(x)$, then $\Gamma' \vdash e : \tau \mid \epsilon$.

The proofs of these lemmas from [38] carry over directly to our system. However, to show subject reduction, we require an extra lemma to reason about state effects.

Lemma 5. (*Stateful effects*)
If $\Gamma \vdash e : \tau \mid \langle st\langle h\rangle \mid \epsilon\rangle$ and $\Gamma \vdash R[e] : \tau' \mid \epsilon'$ then $st\langle h\rangle \in \epsilon'$.

The above lemma essentially states that a stateful effect cannot be discarded in an R context. Later we will generalize this lemma to arbitrary contexts and effects but for subject reduction this lemma is strong enough.

Proof. (Lemma 5) We proceed by induction on the structure of R:
Case $R = [\,]$: By definition $st\langle h\rangle \in \langle st\langle h\rangle \mid \epsilon\rangle$.
Case $R = R'\;e_2$: We have $\Gamma \vdash (R'[e])\;e_2 : \tau' \mid \epsilon'$ and by (APP) we have $\Gamma \vdash R'[e] : \tau_2 \to \epsilon'\;\tau' \mid \epsilon'$. By induction, $st\langle h\rangle \in \epsilon'$.
Case $R = v\;R'$: Similar to previous case.
Case $R = \mathsf{let}\;x \;=\; R'\;\mathsf{in}\;e_2$: By (LET) we have $\Gamma \vdash R'[e] : \tau_1 \mid \langle\rangle$ but that contradicts our assumption.

Case $R = \mathsf{catch}\, R'\, e_2$: By (CATCH) we have $\Gamma \vdash \mathsf{catch}\, R'[e]\, e_2 : \tau' \mid \epsilon'$ where $\Gamma \vdash R'[e] : \tau' \mid \langle exn \mid \epsilon' \rangle$. By induction $st\langle h \rangle \in \langle exn \mid \epsilon' \rangle$ which implies that $st\langle h \rangle \in \epsilon'$.

Now we are ready to prove the subject reduction theorem:

Proof. (Lemma 2) We prove this by induction on the reduction rules of $\longrightarrow$. We will not repeat all cases here and refer to [38], but instead concentrate on the interesting cases, especially with regard to state isolation.
Case $\mathsf{let}\, x = v\, \mathsf{in}\, e \longrightarrow [x \mapsto v]e$: From (LET) we have $\Gamma \vdash v : \tau' \mid \langle\rangle$ and $\Gamma, x : \mathsf{gen}(\Gamma, \tau') \vdash e : \tau \mid \epsilon$. By definition, $\mathsf{gen}(\Gamma, \tau') = \forall \overline{\alpha}.\, \tau'$ where $\overline{\alpha} \notin \mathsf{ftv}(\Gamma)$ and by Lemma 3 we have $\Gamma \vdash [x \mapsto v]e : \tau \mid \epsilon$.
Case $R[\mathsf{hp}\, \varphi.\, e] \longrightarrow \mathsf{hp}\, \varphi.\, R[e]$: This is case is proven by induction over the structure of R:
case $R = []$: Does not apply due to the side condition on $\longrightarrow$.
case $R = R'\, e'$: Then $\Gamma \vdash R'[\mathsf{hp}\, \varphi.\, e]\, e' : \tau \mid \epsilon$ and by (APP) we have $\Gamma \vdash R'[\mathsf{hp}\, \varphi.\, e] : \tau_2 - > \epsilon\, \tau \mid \epsilon$ **(1)** and $\Gamma \vdash e' : \tau_2 \mid \epsilon$ **(2)**. By the induction hypothesis and (1), we have $\Gamma \vdash \mathsf{hp}\, \varphi.\, R'[e] : \tau_2 \rightarrow \epsilon\, \tau \mid \epsilon$. Then by (HEAP) we know $\Gamma, \overline{\varphi}_h \vdash v_j : \tau_j \mid \langle\rangle$ **(3)** and $\Gamma, \overline{\varphi}_h \vdash R'[e] : \tau_2 \rightarrow \epsilon\, \tau \mid \epsilon$ **(4)** where $\varphi = \langle r_1 \mapsto v_1, ..., r_n \mapsto v_n \rangle$. Since $r_1, ..., r_n \notin \mathsf{fv}(e')$ we can use (2) and 4 to conclude $\Gamma, \overline{\varphi} \vdash e' : \tau_2 \mid \epsilon$ **(5)**. Using (APP) with (4) and (5), we have $\Gamma, \overline{\varphi} \vdash R'[e]\, e' : \tau \mid \epsilon$ where we can use (HEAP) with (3) to conclude $\Gamma \vdash \mathsf{hp}\, \varphi.\, R'[e]\, e' : \tau \mid \epsilon$.
case $R = v\, R'$: Similar to the previous case.
case $R = \mathsf{let}\, x = R'\, \mathsf{in}\, e'$: If this is well-typed, then by rule (LET) we must have $\Gamma \vdash R'[\mathsf{hp}\, \varphi.\, e] : \tau' \mid \langle\rangle$. However, due to 5 and (HEAP), we have $st\langle h \rangle \in \langle\rangle$ which is a contradiction. Note that this case is essential, as it prevents generalization of stateful references. For ML, this is also the tricky proof case that only works if one defines special ‘imperative type variables’ [38] or the value restriction. In our case the effect system ensures safety.
Case $\mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, \lambda x.\, e) \longrightarrow \lambda x.\, \mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, e)$: By rule (RUN) and (HEAP) we have that $\Gamma \vdash \lambda x.\, e : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ where $h \notin \mathsf{ftv}(\Gamma, \tau, \epsilon)$ **(1)**. Applying (LAM) gives $\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \epsilon_2$ with $\tau = \tau_1 \!\rightarrow\! \epsilon_2\, \tau_2$. Using (EXTEND) we can also derive $\Gamma, x : \tau_1 \vdash e : \tau_2 \mid \langle st\langle h \rangle \mid \epsilon_2 \rangle$. Due to (1) and $h \notin \tau_1$, we can apply (RUN) and (HEAP) again to infer $\Gamma, x : \tau_1 \vdash \mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, e) : \tau_2 \mid \epsilon_2$ and finally (LAM) again to conclude $\Gamma \vdash \lambda x.\, (\mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, e)) : \tau \mid \epsilon$.
Case $\mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, \mathsf{catch}\, e) \longrightarrow \mathsf{catch}\, (\mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, e))$: Similar to the previous case.
Case $\mathsf{run}\, ([\mathsf{hp}\, \varphi.\,]\, w) \longrightarrow w$ with $\mathsf{frv}(w) \not\cap \mathsf{dom}(\varphi)$ **(1)**: By rule (RUN) and (HEAP) we have that $\Gamma, \overline{\varphi}_h \vdash w : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ where $h \notin \mathsf{ftv}(\Gamma, \tau, \epsilon)$ **(2)**. By (1) it must also be that $\Gamma \vdash w : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ **(3)** (this follows directly if there was no heap binding $\mathsf{hp}\, \varphi.$). We proceed over the structure of w:
case $w = \mathsf{throw}\, v$: Then by (3) we have $\Gamma \vdash \mathsf{throw}\, v : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$, but also $\Gamma \vdash \mathsf{throw}\, v : \tau \mid \epsilon$ since we can choose the result effect freely in (THROW).
case $w = r$: By (VAR) and (3), we have $\Gamma \vdash r : \mathit{ref}\langle h', \tau' \rangle \mid \langle st\langle h \rangle \mid \epsilon \rangle$. where $h \neq h'$ satisfying (2). Since the result effect is free in (VAR), we can also derive $\Gamma \vdash r : \mathit{ref}\langle h', \tau' \rangle \mid \epsilon$
case $w = (r :=)$: As the previous case.
case $w = x$: By (VAR) and (3), we have $\Gamma \vdash x : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ but in (VAR) the result effect is free, so we can also derive $\Gamma \vdash x : \tau \mid \epsilon$.
case other: Similarly.

6.2. Faulty expressions

The main purpose of type checking is of course to guarantee that certain bad expressions cannot occur. Apart from the usual errors, like adding a number to a string, we particularly would like to avoid state errors. There are two aspects to this. One of them is notorious where polymorphic types in combination with state can be unsound (which is not the case in our system because of Lemma 2). But in addition, we would like to show that in our system it is not possible to read or write to locations outside the local heap (encapsulated by run), nor is it possible to let local references escape. To make this precise, the *faulty* expressions are defined as:

- Undefined: $c\, v$ where $\delta(c, v)$ is not defined.
- Escaping read: $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, R[!r])$ where $r \notin \mathsf{dom}(\varphi)$.
- Escaping write: $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, R[r := v])$ where $r \notin \mathsf{dom}(\varphi)$.
- Escaping reference: $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, w)$ where $\mathsf{frv}(w) \cap \mathsf{dom}(\varphi) \neq \varnothing$.
- Not a function: $v\, e$ where v is not a constant or lambda expression.
- Not a reference: $!v$ or $v := e$ where v is not a reference.
- Not an exception: $\mathsf{throw}\, v$ where v is not the unit value.

Lemma 6. (*Faulty expresion are untypable*)
If an expression e is faulty, it cannot be typed, i.e. there exists no Γ, τ, and ϵ such that $\Gamma \vdash e : \tau \mid \epsilon$.

Proof. (Lemma 6) Each faulty expression is handled separately. We show here the interesting cases for escaping reads, writes, and references:
Case $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, R[!r])$ with $r \notin \mathsf{dom}(\varphi)$ **(1)**: To be typed in a context Γ we apply (RUN) and (HEAP) and need to show $\Gamma, \overline{\varphi}_h \vdash R[!r] : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ **(2)**, where $h \notin \mathsf{ftv}(\Gamma, \tau, \epsilon)$ **(3)**. For $R[!r]$ to be well-typed, we also need $\Gamma, \overline{\varphi}_h \vdash !r : \tau' \mid \langle st\langle h' \rangle \mid \epsilon' \rangle$ **(4)** where $\Gamma, \overline{\varphi}_h \vdash r : \mathit{ref}\langle h', \tau' \rangle \mid \langle st\langle h' \rangle \mid \epsilon' \rangle$ **(5)**. From Lemma 5, (4), and (2), it must be that $h' = h$ **(6)**. But since $r \notin \mathsf{dom}(\varphi)$ (1), it follows by (5) and (6), that $\Gamma \vdash r : \mathit{ref}\langle h, \tau' \rangle \mid \langle st\langle h \rangle \mid \epsilon' \rangle$. But that means $h \in \mathsf{ftv}(\Gamma)$ contradicting (3).
Case $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, R[(r :=)])$ with $r \notin \mathsf{dom}(\varphi)$: Similar to the previous case.
Case $\mathsf{run}\, (\mathsf{hp}\, \varphi.\, w)$ where $\mathsf{frv}(w) \cap \mathsf{dom}(\varphi) \neq \varnothing$ **(1)** To be typed in a context Γ we need to show by (HEAP) and (RUN) that $\Gamma, \overline{\varphi}_h \vdash w : \tau \mid \langle st\langle h \rangle \mid \epsilon \rangle$ where $h \notin \mathsf{ftv}(\Gamma, \tau, \epsilon)$ **(2)**. If $w = \mathsf{throw}\, v$ then by (THROW) the type of v is $()$ and thus v is the unit constant. But $\mathsf{frv}(()) = \varnothing$ contradicting our assumption. Otherwise, $w = b$ and cannot contain an arbitrary e. Since $\mathsf{frv}(w) \neq \varnothing$ (1), it must be that w is either one of r or $(r :=)$ with $r \in \mathsf{dom}(\varphi)$. To be well-typed, $\Gamma, \overline{\varphi}_h \vdash r : \mathit{ref}\langle h, \tau' \rangle \mid \epsilon'$ must hold. However, the possible types for r and $(r :=)$ are $\mathit{ref}\langle h, \tau' \rangle$ and $\tau' \rightarrow st\langle h \rangle\, ()$ and in both cases $h \in \mathsf{ftv}(\tau)$ which contradicts (2).

7. Effectful semantics

Up till now, we have used the effect types to good effect and showed that our system is semantically sound, even though state and polymorphic types are notoriously tricky to combine. Moreover, we showed that local state isolation through run is sound and statically prevents references from escaping. In essence, this is a combination of the semantics of Core ML by Wright and Felleisen [38] and the proof of heap safety of the state-monad by Launchbury and Sabry [13] (even though our formalization is quite different).

But the true power of the effect system is really to enable more reasoning about the behavior of a program at a higher level. In particular, the absence of certain effects determines the absence of certain answers. For example, if the exception effect is not inferred, then evaluating the program will never produce an answer of the form $\mathsf{throw}\, v$ or $\mathsf{hp}\, \varphi.\, \mathsf{throw}\, v$! It would not be entirely correct to say that such program never throws an exception: indeed, a local catch block can handle such exceptions. The right answer is that if such program throws an exception, then it is guaranteed by the type that all of those exceptions are handled. We can state the exception property formally as:

Theorem 6. (*Exceptions*)
If $\Gamma \vdash e : \tau \mid \epsilon$ where $exn \notin \epsilon$ then either $e \Uparrow$, $e \longmapsto\!\!\!\!\twoheadrightarrow v$ or $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{hp}\,\varphi.\, v$.

Proof. By contradiction over the result terms:
Case $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{throw}\, v$: By subject reduction (Lemma 2), it must be $\Gamma \vdash \mathsf{throw}\, v : \tau \mid \epsilon$. Using the type rule for throw with (APP), it must be the case that $\epsilon \equiv \langle exn \mid \epsilon' \rangle$ contradicting our assumption.
Case $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{hp}\,\varphi.\,\mathsf{throw}\, v$: Similar to the previous case.

Similarly to the exception case, we can state such theorem over heap effects too. In particular, if the $st\langle h\rangle$ effect is absent, then evaluation will not produce an answer that contains a heap, i.e. $\mathsf{hp}\,\varphi.\, w$. Again, it would not be right to say that the program itself never performs any stateful operations. The correct way is to say, that if the program performs any stateful computation, it is guaranteed by the type that such behavior is truly encapsulated inside a run construct and its stateful behavior is not observable from outside. Formally, we can state this as:

Theorem 7. (*State*)
Iff $\Gamma \vdash e : \tau \mid \epsilon$ where $st\langle h\rangle \notin \epsilon$ then either $e \Uparrow$, $e \longmapsto\!\!\!\!\twoheadrightarrow v$ or $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{throw}\, v$.

Proof. Again by contradiction over the result terms:
Case $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{hp}\,\varphi.\, v$: By subject reduction (Lemma 2), it must be $\Gamma \vdash \mathsf{hp}\,\varphi.\, v : \tau \mid \epsilon$. Using (HEAP), it must be the case that $\epsilon \equiv \langle st\langle h\rangle \mid \epsilon' \rangle$ contradicting our assumption.
Case $e \longmapsto\!\!\!\!\twoheadrightarrow \mathsf{hp}\,\varphi.\,\mathsf{throw}\, v$: Similar to the previous case.

Our most powerful theorem is about the divergence effect; in particular, if the divergent effect is absent, then evaluation is guaranteed to terminate!

Theorem 8. (*Divergence*)
If $\Gamma \vdash e : \tau \mid \epsilon$ where $div \notin \epsilon$ then $e \longmapsto\!\!\!\!\twoheadrightarrow a$.

The proof of this lemma is more complicated as we cannot use subject reduction to show this by contradiction. Instead, we need to do this proof using induction over logical relations [8].

In our case, we say that if $\cdot \vdash e : \tau \mid \epsilon$, then e is in the set $\mathcal{R}(\tau \mid \epsilon)$, "the reduceable terms of type τ with effect ϵ", if $div \notin \epsilon$ and (1) when τ is a non-arrow type, if e halts, and (2) when $\tau = \tau_1 \rightarrow \epsilon_2\, \tau_2$, if e halts and if for all $e_1 \in \mathcal{R}(\tau_1 \mid \epsilon)$, we have that $e\, e_1 \in \mathcal{R}(\tau_2 \mid \epsilon_2)$.

The proof of Theorem 8 is a standard result [8] and is a corollary from the following two main lemmas:

Lemma 7. ($\mathcal{R}$ *is preserved by reduction*) Iff $\cdot \vdash e : \tau \mid \epsilon$, $e \in \mathcal{R}(\tau \mid \epsilon)$, and $e \longmapsto e'$, then also $e' \in \mathcal{R}(\tau \mid \epsilon)$.

Lemma 8. (*A well-typed term is in* $\mathcal{R}$) If $\cdot \vdash e : \tau \mid \epsilon$ and $div \notin \epsilon$, then $e \in \mathcal{R}(\epsilon \mid \tau)$.

Proof. (Lemma 7) This is shown by induction over the type τ. For atomic types, this holds by definition. For arrow types, $\tau_1 \rightarrow \epsilon_2\, \tau_2$ we must show for a given $e_1 \in \mathcal{R}(\tau_1 \mid \epsilon)$ that if $e\, e_1 \in \mathcal{R}(\tau_2 \mid \epsilon_2)$, then also $e'\, e_1 \in \mathcal{R}(\tau_2 \mid \epsilon_2)$ (and the other direction). By (APP), it must be $\epsilon_2 = \epsilon$ **(1)**. We can now proceed over the structure of reductions on $e\, e_1$:
Case $(!)\, e_1$: In this case, since (READ) has a div effect, we have by (1) that $div \in \epsilon$ contradicting our assumption. Note that if we would have cheated and not include div in the type, we would have gotten a reduction to some v which we could not show to be strongly normalizing, and thus if it is an element of $\mathcal{R}(\tau_2 \mid \epsilon_2)$.
Case $\mathsf{fix}\, e_1$: As the previous case.
Case $(\lambda x.\, e_2)\, e_1$: In this case, we can reduce to $e'\, e_1$, and by the induction hypothesis $e'\, e_1 \in \mathcal{R}(\tau \mid \epsilon_2)$ since τ_2 is smaller.

Proof. (Lemma 8) This is proven over the structure of the type derivation. However, as usual, we need to strengthen our induction hypothesis to include the environment. We extend $\mathcal{R}$ over environments to be a set of substitutions:

$$\mathcal{R}(\Gamma) = \{\theta \mid \mathsf{dom}(\Gamma) = \mathsf{dom}(\theta) \wedge \forall (x : \tau \in \Gamma).\, \theta x \in \mathcal{R}(\tau \mid \langle\rangle)\}$$

where we assume a monomorphic environment for simplicity but we can extend this easily to a (first-order) polymorphic setting. Our strengthened lemma we use for our proof is:

$$\text{if } \Gamma \vdash e : \tau \mid \epsilon \wedge \theta \in \mathcal{R}(\Gamma) \wedge div \notin \epsilon \text{ then } \theta e \in \mathcal{R}(\tau \mid \epsilon)$$

The induction is standard, and we show only some sample cases:
Case (FIX): Since the result effect is free, we can choose any ϵ such that $div \notin \epsilon$. Indeed, just an occurrence of fix is ok – only an application may diverge.
Case (APP): By the induction hypothesis and (APP) we have $\theta e_1 \in \mathcal{R}(\tau_2 \rightarrow \epsilon\, \tau \mid \epsilon)$ and $\theta e_2 \in \mathcal{R}(\tau_2 \mid \epsilon)$. By definition of $\mathcal{R}(\tau_2 \rightarrow \epsilon\, \tau \mid \epsilon)$, $\theta e_1\, \theta e_2 \in \mathcal{R}(\tau_2 \mid \epsilon)$ and therefore $\theta(e_1\, e_2) \in \mathcal{R}(\tau_2 \mid \epsilon)$. Note that the induction hypothesis ensures that $div \notin \epsilon$ and therefore we cannot apply a potentially divergent function (like fix or $(!)$).
Case other: Standard, except that for effect elimination rules, we need to show that div is not eliminated.

8. Related work

A main contribution of this paper is showing that our notion of mutable state is sound, in particular the combination of mutable state and polymorphic let- bindings is tricky as shown by Tofte [33] for the ML language. Later, variants of the ML value restriction are studied by Leroy [16].

Safe state encapsulation using a lazy state monad was first proven formally by Launchbury and Sabry [13]. Their formalization is quite different though from ours and applies to a lazy store in a monadic setting. In particular, in their formalization there is no separate heap binding, but heaps are always bound at the outer *run*. We tried this, but it proved difficult in our setting; for example, it is hard to state the stateful lemma since answers would never contain an explicit heap. Very similar to our state encapsulation is region inference [34]. Our *run* operation essentially delimits a heap region. Regions live at the value level though, and we can for example not access references in several regions at once.

Independently of our work, Lindley and Cheney [17] also used row polymorphism for effect types. Their approach is based on presence/absence flags [25] to give effect types to database operations in the context of the Links web programming language. The main effects of the database operations are *wild*, *tame*, and *hear*, for arbitrary effects including divergence, pure queries, and asynchronous messages respectively.They report on practical experience exposing effect types to the programmer and discuss various syntax forms to easily denote effect types.

The problems with arbitrary effects have been widely recognized, 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 [7, 18] which was later extended by Talpin [31] and others [30, 22]. These systems are closely related since they describe polymorphic effect systems and use type constraints to give principal types. The system described by Nielson *et al.* [22] also requires the effects to form a complete lattice with meets and joins.

Java contains a simple effect system where each method is labeled with the exceptions it might raise [9]. A system for finding uncaught exceptions was developed for ML by Pessaux *et al.* [23]. A more powerful system for tracking effects was developed by Benton [2] who also studies the semantics of such effect systems [3]. Recent work on effects in Scala [27] shows how even a restricted form of polymorphic effect types can be used to track effects for many programs in practice.

Tolmach [35] describes an effect analysis for ML in terms of effect monads, 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 [21].

Marino *et al.* recently produced a generic type-and-effect system [19]. 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 but has no semantics on its own. For example, it would be sound for the effect system to have "+" grant the $canThrow$ privilege to its arguments, and one has to do an additional proof to show that the effects in these systems actually correspond to an intended meaning.

Wadler and Thiemann showed the close relation between effect systems and monads [36] and showed how any effect system can be translated to a monadic version. For our particular system though a monadic translation is quite involved due to polymorphic effects; essentially we need dependently typed operations and we leave a proper monadic semantics for future work.

References

[1] Andreas Abel. foetus termination checker for simple functional programs. unpublished note, 1998.

[2] Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *TLDI '07: Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 15–26, 2007.

[3] Nick Benton, Andrew Kennedy, Lennart Beringer, and Martin Hofmann. Relational semantics for effect-based program transformations with dynamic allocation. In *PPDP '07: Proc. of the 9th ACM SIGPLAN int. conf. on Principles and Practice of Declarative Prog.*, pages 87–96, 2007.

[4] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *9th82*, pages 207–212, 1982.

[5] Jean-Christophe Filliâtre. A Functional Implementation of the Garsia–Wachs Algorithm. In *ACM SIGPLAN Workshop on ML*, Victoria, British Columbia, Canada, September 2008. ACM.

[6] Ben R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, 1996.

[7] David K. Gifford and John M. Lucassen. Integrating functional and imperative programming. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 28–38, 1986.

[8] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types*. Cambridge University Press, 1989.

[9] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

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

[11] Mark P. Jones. A theory of qualified types. In *4th. European Symposium on Programming (ESOP'92)*, volume 582 of *Lecture Notes in Computer Science*, pages 287–306. Springer-Verlag, February 1992.

[12] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995.

[13] John Launchbury and Amr Sabry. Monadic state: Axiomatization and type safety. In *ICFP'97*, pages 227–238, 1997.

[14] Daan Leijen. Extensible records with scoped labels. In *Proc. of the 2005 Symp.f on Trends in Functional Programming (TFP'05)*, September 2005.

[15] Daan Leijen. Try Koka online. http://rise4fun.com/koka/tutorial and http://koka.codeplex.com, 2012.

[16] Xavier Leroy. Polymorphism by name for references and continuations. In *POPL '93: Proc. of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 220–231, 1993.

[17] Sam Lindley and James Cheney. Row-based effect types for database integration. In *TLDI'12*, pages 91–102, 2012.

[18] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *POPL '88*, pages 47–57, 1988.

[19] Daniel Marino and Todd Millstein. A generic type-and-effect system. In *TLDI '09: Proceedings of the 4th international workshop on Types in language design and implementation*, pages 39–50, 2009.

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

[21] David A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007.

[22] Hanne Riis Nielson, Flemming Nielson, and Torben Amtoft. Polymorphic subtyping for effect analysis: The static semantics. In *Selected papers from the 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages*, pages 141–171, 1997.

[23] François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. In *POPL '99*, pages 276–290, 1999.

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

[25] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[26] Didier Remy. Programming objects with ML-ART, an extension to ML with abstract and record types. In *TACS '94: Proc. Int. Conf. on Theoretical Aspects of Computer Software*, pages 321–346, 1994.

[27] Lukas Rytz, Martin Odersky, and Philipp Haller. Lightweight polymorphic effects. In *European Conference on Object-Oriented Programming (ECOOP), Beijing, China*, June 2012.

[28] Martin Sulzmann. Designing record systems. Technical Report YALEU/DCS/RR-1128, Yale University, April 1997.

[29] Martin Sulzmann. Type systems for records revisited. Unpublished report, June 1998.

[30] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Inf. Comput.*, 111(2):245–296, 1994.

[31] 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.

[32] Ross Tate and Daan Leijen. Convenient explicit effects using type inference with subeffects. Technical Report MSR-TR-2010-80, Microsoft Research, June 2010.

[33] Mads Tofte. Type inference for polymorphic references. *Inf. Comput.*, 89(1):1–34, September 1990.

[34] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):724–767, 1998.

[35] Andrew P. Tolmach. Optimizing ML using a hierarchy of monadic types. In *TIC '98*, pages 97–115, 1998.

[36] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, 2003.

[37] Mitchell Wand. Complete type inference for simple objects. In *Proceedings of the 2nd. IEEE Symposium on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in LICS'88, page 132.

[38] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994.