No value restriction is needed for algebraic effects and handlers

Ohad Kammar[∴][†]
University of Cambridge Computer Laboratory and
University of Oxford Department of Computer Science, England
and
Matija Pretnar[†]
Faculty of Mathematics and Physics, University of Ljubljana, Slovenia

Abstract

We present a straightforward, sound, Hindley-Milner polymorphic type system for algebraic effects and handlers in a call-by-value calculus, which, to our surprise, allows type variable generalisation of arbitrary computations, and not just values. We first recall that the soundness of unrestricted call-by-value Hindley-Milner polymorphism is known to fail in the presence of computational effects such as reference cells and continuations, and that many programming examples can be recast to use effect handlers instead of these effects. After presenting the calculus and its soundness proof, formalised in Twelf, we analyse the expressive power of effect handlers with respect to state effects. We conjecture handlers alone cannot express reference cells, but show they can simulate dynamically scoped state, establishing that dynamic binding also does not need a value restriction.

1 Introduction

The following *OCaml* example (Leroy, 1992) demonstrates the problematic interaction between Hindley-Milner polymorphism, which increases code reuse, and computational effects, such as reference cells, in a call-by-value language:

$$\begin{array}{ll}
\mathbf{let}\ r = \ \mathbf{ref}\ [\,]\ \mathbf{in} & (*\ \text{generalise}\ r : \forall\alpha.\alpha\ \mathsf{list\ ref}\ *) \\
r := [()]; & (*\ \text{specialise}\ \alpha := \mathsf{unit}\ *) \\
\mathbf{true} :: !r & (*\ \text{specialise}\ \alpha := \mathsf{bool}\ *)
\end{array}$$

A naïve type inference algorithm would assign the type α $\mathsf{list\ ref}$ to the term $\mathbf{ref}\ [\,]$. Unrestricted, it would assign to r the *type scheme* $\forall\alpha.\alpha\ \mathsf{list\ ref}$. But doing so allows us to instantiate r with the unit type $\alpha := \mathsf{unit}$ to store the singleton list with the unit value,

∴ Supported by the European Research Council grant 'events causality and symmetry — the next-generation semantics', and the Engineering and Physical Sciences Research Council grant EP/N007387/1 'quantum computing as a programming language'.
† The material is based upon work supported by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF under Award No. FA9550-14-1-0096.

and then to instantiate r with the boolean type $\alpha :=$ bool. The result is a list whose second element is the unit value, but appears to the type system as a list of booleans.

The current way to avoid this well-known unsound behaviour (Pierce, 2002; Harper & Lillibridge, 1993b; Rémy, 2015) is to enforce a *value restriction*: the inference algorithm will generalise the type variables only in value terms that cannot be reduced further (Wright, 1995). While this restriction can be weakened to allow some computation (Garrigue, 2004), it still rules out sound pure programs:

$$\begin{array}{ll} \textbf{let}\, id = (\textbf{fun}\, f \mapsto f)\ (\textbf{fun}\, x \mapsto x)\ \textbf{in} & (*\ \text{id is } \textit{not} \text{ polymorphic } *) \\ id\,(id) & (*\ \text{type error } *) \end{array}$$

The problem only arises when all three components are present: computational effects, polymorphism, and call-by-value evaluation order. Without effects, Milner's original calculus soundly integrates call-by-value with type inference (Milner, 1978). Without polymorphism, computational effects behave predictably in call-by-value languages like *ML*, as opposed to call-by-name languages like Haskell, which require additional features such as monads to make effects predictable. Without call-by-value, Leroy (1993) combines computational effects with polymorphism without restriction. Leroy's language has two constructs for sequencing: a call-by-name polymorphic construct $\textbf{let}\, x = c_1\ \textbf{in}\ c_2$ in which c_1 is re-executed whenever it is specialised in c_2, and a call-by-value monomorphic construct $\textbf{do}\, x \leftarrow c_1\ \textbf{in}\ c_2$ in which c_1 is only evaluated once, but its type is not generalised. The situation is identical in the *Haskell* programming language, from which we borrow this notation.

Programming with algebraic effects and handlers (Bauer & Pretnar, 2015) is a new approach to structuring functional programs with computational effects. The programmer declares a collection of *algebraic effect operations* with which she structures her effectful code. Then, separately, she defines *effect handlers* that implement these abstract operations. Bauer & Pretnar's *Eff* programming language is a strict (i.e., call-by-value) functional language with Hindley-Milner polymorphism, in which all computational effects are treated as algebraic effects that can be handled. As *Eff* combines the three problematic components (strictness, polymorphism, effects), it currently imposes the standard value restriction on the programmer.

In this paper, we show that if only algebraic effects and handlers are present, no value restriction is necessary. We present a straightforward sound Hindley-Milner polymorphic type system for a call-by-value language that incorporates computational effects in the form of algebraic effects and their handlers. In the given language, we can assign a polymorphic type to x in $\textbf{do}\, x \leftarrow c_1\ \textbf{in}\ c_2$ not only if c_1 is a pure computation, like in the *id* example above, but also if c_1 calls effects. Keep in mind that the language is strict, so c_1 gets evaluated only once.

In order to simplify the presentation, we present a type system without its associated complete inference algorithm. Doing so decouples the algorithmic concerns of finding principal types and its complexity from the semantic concern for soundness. As first-class polymorphism typically makes type inference undecidable (Wells, 1999), our type system uses *ML*-style polymorphism.

An important point of difference between our calculus and Bauer & Pretnar's *Eff* is the treatment of *effect instances* (Bauer & Pretnar, 2015, 2014; Pretnar, 2014). Instances provide dynamic generation of effect names, increasing the modularity of effectful code. We do not know how to combine instances with polymorphism, and so we do not advocate to lift the value restriction from *Eff*.

The rest of the paper is structured as follows. In Sec. 2 we review handlers as a programming abstraction through an idealised core calculus of algebraic effects and handlers, and demonstrate its use by simulating global state. In Sec. 3 we give a type-and-effect system to the core calculus and sketch the proof of its soundness. We formalized the proof in the Twelf proof assistant (Pfenning & Schürmann, 1999), extending Bauer & Pretnar's (2014) existing formalization of *Eff*'s core calculus[1] In Sec. 4 we evaluate our type system and discuss its expressiveness with respect to mutable references and dynamically scoped state. In Sec. 5 we summarise and elaborate on related work. In Sec. 6 we conclude.

2 Handlers of algebraic effects

Algebraic effects are an approach to computational effects based on a premise that impure behaviour arises from a set of *operations* such as get and set for mutable store, read and print for interactive input and output, or raise for exceptions (Plotkin & Power, 2003). This approach naturally gives rise to *handlers* not only of exceptions, but of any other effect, yielding a novel programming abstraction that, amongst others, can capture backtracking, co-operative multi-threading, Unix-style stream redirection, and delimited continuations (Plotkin & Pretnar, 2013; Bauer & Pretnar, 2015).

2.1 Language

We base our development on the calculus (Fig. 1) given in Pretnar's (2015) tutorial. The language is a variant of the fine-grained call-by-value λ-calculus of Levy *et al.* (2003), in which terms are split into inert *values* and potentially effectful *computations*.

Programmers introduce effects with the construct $\mathsf{op}(v; y.\, c)$, which calls the operation op with the parameter v. The effect invocation may yield a value to the continuation c using the bound variable y. Programmers define the meaning of such operation calls by enclosing them in effect handlers. A handler specifies a return clause, used when the computation returns a final value, and a collection of operation clauses $\mathsf{op}(x; k) \mapsto c$, which specify how we should execute an invocation of the operation op called with the parameter x and a continuation k. The underlying idea is that operation calls behave as signals that propagate outwards until they reach a handler with a matching clause.

Our handlers are *deep*: any additional effects in the continuation are also handled by the current handler. Our handlers are also *forwarding*: unhandled operations propagate through each handler until they are handled or reach the top level. None of these design choices is essential to the development below, but we make them to mirror *Eff*'s design choices.

We use the following syntactic sugar (Fig. 1): semicolons elaborate to binding fresh (dummy) variables; we use sequencing, with appropriate freshly bound variables, to allow

[1] Accessible from the Oxford Research Archive (DOI: `10.5287/bodleian:KZk7zaXZb`).

Syntax

$v ::=$		value
	x	variable
$\mid$	$\mathbf{true} \mid \mathbf{false}$	boolean constants
$\mid$	$\mathbf{fun}\, x \mapsto c$	function
$\mid$	h	handler
$h ::=$		handler
	$\mathbf{handler}\, \{\mathbf{return}\, x \mapsto c_r,$	return clause
	$\mathsf{op}_1(x_1;k_1) \mapsto c_1, \dots, \mathsf{op}_n(x_n;k_n) \mapsto c_n\}$	operation clauses
$c ::=$		computation
	$\mathbf{return}\, v$	return
$\mid$	$\mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2$	sequencing
$\mid$	$\mathsf{op}(v; y.\, c)$	operation call
$\mid$	$\mathbf{if}\, v \,\mathbf{then}\, c_1 \,\mathbf{else}\, c_2$	conditional
$\mid$	$v_1\, v_2$	application
$\mid$	$\mathbf{with}\, v \,\mathbf{handle}\, c$	handling

Syntactic sugar

Sugar	***Elaboration***
$\_$	fresh variable binding
$c_1; c_2$	$\mathbf{do}\, \_ \leftarrow c_1 \,\mathbf{in}\, c_2$
$v_1\, c_2$	$\mathbf{do}\, a \leftarrow c_2 \,\mathbf{in}\, v_1\, a$
$c_1\, v_2$	$\mathbf{do}\, f \leftarrow c_1 \,\mathbf{in}\, f\, v_2$
$c_1\, c_2$	$\mathbf{do}\, f \leftarrow c_1 \,\mathbf{in}\, \mathbf{do}\, a \leftarrow c_2 \,\mathbf{in}\, f\, a$
$\mathbf{if}\, c \,\mathbf{then}\, c_1 \,\mathbf{else}\, c_2$	$\mathbf{do}\, b \leftarrow c \,\mathbf{in}\, \mathbf{if}\, b \,\mathbf{then}\, c_1 \,\mathbf{else}\, c_2$
$\mathsf{op}(c_p; y.\, c_k)$	$\mathbf{do}\, p \leftarrow c_p \,\mathbf{in}\, \mathsf{op}(p; y.\, c_k)$
$\mathbf{fun}\, x\, y \mapsto c$	$\mathbf{fun}\, x \mapsto \mathbf{return}\, (\mathbf{fun}\, y \mapsto c)$
$\mathsf{op}(c)$	$\mathsf{op}(c; y.\, \mathbf{return}\, y)$

Fig. 1. A calculus for effect handlers

computations in places where values are expected in function calls, conditionals, and operation calls; function introduction may abstract over two arguments; and we assume a trivial continuation in operations without a continuation argument. In our examples, we further assume to have the type unit with the sole inhabitant $()$ and abbreviate $\mathsf{op}(())$ to $\mathsf{op}()$.

2.2 State handlers

We represent state with an operation set, which sets the state contents to a given parameter and returns $()$, and get, which takes a unit parameter and returns the state contents. For example, here is a computation that toggles the state and returns the old value:

$$\begin{aligned}T :=\ & \textbf{if}\ \mathsf{get}()\ \textbf{then}\\ & \quad \mathsf{set}(\textbf{false}); \textbf{return true}\\ & \textbf{else}\\ & \quad \mathsf{set}(\textbf{true}); \textbf{return false}\end{aligned}$$

In the runtime of Bauer & Pretnar's (2015) *Eff*, there is a pre-defined collection of effects that receive special treatment: runtime errors and memory accesses. If these effects are not handled by the program, the runtime will handle them, invoking the corresponding real computational effects. However, in our calculus, the behaviour of operations will be determined exclusively by handlers, and computations such as T get stuck when evaluated without an appropriate enclosing handler.

A simple example of a handler for stateful computations sets the state to a fixed value, say **true**, and ignores all its modifications:

$$\begin{aligned}H_C := \textbf{handler}\ \{ & \mathsf{get}(\_;k) \mapsto k\ \textbf{true},\\ & \mathsf{set}(s\,;k) \mapsto k\ (),\\ & \textbf{return}\ x \mapsto \textbf{return}\ x\}\end{aligned}$$

Whenever we call a get operation, we yield **true** to the continuation, and ignore any set operation calls by yielding the expected unit value $()$ and doing nothing else. The return clause of the handler states that we return values unmodified. Thus, when we handle T with H_C, we get back the result **true**, no matter how many times we previously called T.

A more useful handler is one that handles get and set in a way that results in the expected stateful behaviour. It uses a technique called *parameter-passing* (Plotkin & Pretnar, 2013), where we transform the handled computation into a function that passes around a parameter, in our case the state contents:

$$\begin{aligned}H_{ST} := \textbf{handler}\ \{ & \mathsf{get}(\_;k) \mapsto \textbf{return}\ (\textbf{fun}\ s \mapsto (k\ s)\ s),\\ & \mathsf{set}(s';k) \mapsto \textbf{return}\ (\textbf{fun}\ \_ \mapsto (k\ ())\ s'),\\ & \textbf{return}\ x \mapsto \textbf{return}\ (\textbf{fun}\ \_ \mapsto \textbf{return}\ x)\}\end{aligned}$$

We handle get with a function that takes the current state contents s and in the first application, passes them as a result of get to the continuation. As our handlers are deep, the continuation is further handled into a function, which we again need to supply with the state contents. Since reading does not modify the state, we again pass s. We handle set by first passing the unit result, and then applying the handled continuation to the new state s' as given by the parameter of set. The return clause of H_{ST} also needs to produce a function that depends on the given state, in particular, a function that returns the given value regardless of the state contents.

2.3 Operational semantics

To see how to use H_{ST} to simulate state, consider the operational semantics of the calculus, also copied verbatim from Pretnar's (2015) tutorial. We give the semantics in terms of the

Semantics

$$\frac{c_1 \rightsquigarrow c_1'}{\mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2 \rightsquigarrow \mathbf{do}\, x \leftarrow c_1' \,\mathbf{in}\, c_2} \qquad \frac{}{\mathbf{do}\, x \leftarrow \mathbf{return}\, v \,\mathbf{in}\, c \rightsquigarrow c[v/x]}$$

$$\frac{}{\mathbf{do}\, x \leftarrow \mathsf{op}(v; y.\, c_1) \,\mathbf{in}\, c_2 \rightsquigarrow \mathsf{op}(v; y.\, \mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2)}\,(\textsc{do-op})$$

$$\frac{}{\mathbf{if\ true\ then}\, c_1 \,\mathbf{else}\, c_2 \rightsquigarrow c_1} \qquad \frac{}{\mathbf{if\ false\ then}\, c_1 \,\mathbf{else}\, c_2 \rightsquigarrow c_2}$$

$$\frac{}{(\mathbf{fun}\, x \mapsto c)\, v \rightsquigarrow c[v/x]}$$

For every $h = \mathbf{handler}\, \{\mathbf{return}\, x \mapsto c_r, \mathsf{op}_1(x_1; k_1) \mapsto c_1, \ldots, \mathsf{op}_n(x_n; k_n) \mapsto c_n\}$, define:

$$\frac{c \rightsquigarrow c'}{\mathbf{with}\, h \,\mathbf{handle}\, c \rightsquigarrow \mathbf{with}\, h \,\mathbf{handle}\, c'} \qquad \frac{}{\mathbf{with}\, h \,\mathbf{handle}\, (\mathbf{return}\, v) \rightsquigarrow c_r[v/x]}$$

$$\frac{\mathsf{op}_i \in \{\mathsf{op}_1, \ldots, \mathsf{op}_n\}}{\mathbf{with}\, h \,\mathbf{handle}\, \mathsf{op}_i(v; y.\, c) \rightsquigarrow c_i[v/x_i, (\mathbf{fun}\, y \mapsto \mathbf{with}\, h \,\mathbf{handle}\, c)/k_i]}\,(\textsc{handled-op})$$

$$\frac{\mathsf{op} \notin \{\mathsf{op}_1, \ldots, \mathsf{op}_n\}}{\mathbf{with}\, h \,\mathbf{handle}\, \mathsf{op}(v; y.\, c) \rightsquigarrow \mathsf{op}(v; y.\, \mathbf{with}\, h \,\mathbf{handle}\, c)}\,(\textsc{unhandled-op})$$

Fig. 2. The operational semantics for effect handlers

small-step relation $c \rightsquigarrow c'$, defined in Fig. 2. As expected, there is no such relation for values, as these are inert.

The rules for conditionals and function application are standard. For the sequencing construct, $\mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2$, we start by evaluating c_1. If c_1 returns some value v, we bind it to x and evaluate c_2. But if c_1 calls an operation, we propagate the call outwards and defer further evaluation to the continuation of the call, for example:

$$\begin{aligned}
&\mathbf{do}\, x_1 \leftarrow (\mathbf{do}\, x_2 \leftarrow \mathsf{op}(v; y.\, c_2) \,\mathbf{in}\, c_1) \,\mathbf{in}\, c && \rightsquigarrow \\
&\mathbf{do}\, x_1 \leftarrow \mathsf{op}(v; y.\, \mathbf{do}\, x_2 \leftarrow c_2 \,\mathbf{in}\, c_1) \,\mathbf{in}\, c && \rightsquigarrow \\
&\mathsf{op}(v; y.\, \mathbf{do}\, x_1 \leftarrow (\mathbf{do}\, x_2 \leftarrow c_2 \,\mathbf{in}\, c_1) \,\mathbf{in}\, c)
\end{aligned}$$

In our account, we gloss over the standard issues with capture-avoiding substitution and implicitly assume the appropriate freshness conditions. For example, in this case, that y is fresh for c and c_1.

To evaluate **with** h **handle** c, we start by evaluating c. If it returns a value, we continue by evaluating the return clause of h. If c calls an operation op, there are two options. If h has a matching clause for op, we start evaluating this clause, passing in the parameter and the continuation. Recall that our handlers are deep, thus the continuation k is also handled by the current handler, see HANDLED-OP. If h does not have a matching clause, we forward the call outwards just like in sequencing, see UNHANDLED-OP.

Let us return to the state handler H_{ST}. If we use it on a stateful computation, no effects occur as the handled computation returns a function waiting for an initial state. To run it, we need to apply this function to the initial state. We abbreviate such an application by:

$$\langle c, s\rangle := (\mathbf{with}\ H_{ST}\ \mathbf{handle}\ c)\ s$$

Note how we use the syntactic sugar for call-by-value function calls from Fig. 1.

Even though our calculus is pure, we can show the handler H_{ST} simulates global state in the following way. Let $\overset{st}{\rightsquigarrow}$ be the usual small-step semantics for global state, i.e.:

$$\langle \mathsf{get}(), s\rangle \overset{st}{\rightsquigarrow} \langle \mathbf{return}\ s, s\rangle \qquad \langle \mathsf{set}(s'), s\rangle \overset{st}{\rightsquigarrow} \langle \mathbf{return}\ (), s'\rangle$$

$$\frac{\langle c_1, s\rangle \overset{st}{\rightsquigarrow} \langle c_1', s'\rangle}{\langle \mathbf{do}\ x \leftarrow c_1\ \mathbf{in}\ c_2, s\rangle \overset{st}{\rightsquigarrow} \langle \mathbf{do}\ x \leftarrow c_1'\ \mathbf{in}\ c_2, s'\rangle}$$

and so on.

We can prove that each transition $\langle c_1, s\rangle \overset{st}{\rightsquigarrow} \langle c_1', s'\rangle$ has a matching sequence of transitions $\langle c_1, s\rangle \rightsquigarrow^{+} \langle c_1', s'\rangle$, and therefore the handler semantics simulates the operational semantics for global state. First, calculate:

$$\begin{aligned}
\langle \mathsf{get}(), s\rangle &= (\mathbf{with}\ H_{ST}\ \mathbf{handle}\ (\mathsf{get}((); y.\,\mathbf{return}\ y)))\ s\\
&\rightsquigarrow (\mathbf{fun}\ s' \mapsto ((\mathbf{fun}\ y \mapsto \mathbf{with}\ H_{ST}\ \mathbf{handle}\ (\mathbf{return}\ y))\ s')\ s')\ s\\
&\rightsquigarrow ((\mathbf{fun}\ y \mapsto \mathbf{with}\ H_{ST}\ \mathbf{handle}\ (\mathbf{return}\ y))\ s)\ s\\
&\rightsquigarrow (\mathbf{with}\ H_{ST}\ \mathbf{handle}\ (\mathbf{return}\ s))\ s\\
&= \langle \mathbf{return}\ s, s\rangle
\end{aligned}$$

Similarly, we can prove:

$$\langle \mathsf{set}(s'), s\rangle \rightsquigarrow^{+} \langle \mathbf{return}\ (), s'\rangle$$

We then conclude by straightforward induction on the relation $\langle c_1, s\rangle \overset{st}{\rightsquigarrow} \langle c_1', s'\rangle$.

In summary, the H_{ST} handler faithfully simulates state. For more details on simulating state, see Bauer & Pretnar (2014) and Danvy (2006). Therefore, even though our calculus is pure, it faithfully simulates impure computation. By giving an unrestricted Hindley-Milner type system to this calculus, we now show that the effects expressible by effect handlers interact well with polymorphism.

3 Type system

The type-and-effect system (Figs. 3–4) closely follows Pretnar (2015). It comprises two kinds of types: values have simple types A, while computation types are additionally annotated with finite sets of operations Σ, as in the effect system of Lucassen & Gifford (1988).

We modify Pretnar's system in two ways. The first modification is minor. We generalise the type system to allow for more flexible *local* operation signatures Σ, where operations may have different types when handled by different handlers, as in Kammar *et al.* (2013). In contrast, Pretnar's account posits a global assignment of predefined types to the effect operations, and the effect annotations Σ only list which operations may be present. Local signatures allow the same operation symbol to appear in disjoint parts of the program with

Types

A, B ::=		value type
	α	type variable
	$\mid$ bool	boolean type
	$\mid$ $A \to \underline{C}$	function type
	$\mid$ $\underline{C} \Rightarrow \underline{D}$	handler type
$\underline{C}, \underline{D}$ ::=	$A\,!\,\Sigma$	computation type
	$\forall \vec{\alpha}.A$	scheme
Σ	::= $\{\mathsf{op}_1 : A_1 \to B_1, \ldots, \mathsf{op}_n : A_n \to B_n\}$	effect signature
Θ	::= $\{\alpha_1, \ldots, \alpha_n\}$	type variable environment
Γ	::= $\varnothing \mid \Gamma, x : A$	monomorphic environment
Ξ	::= $\varnothing \mid \Xi, x : \forall \vec{\alpha}.A$	polymorphic environment

Fig. 3. Polymorphic types and effects for effect handlers

different types. Local signatures also give the calculus stronger theoretical properties, such as strong normalisation and simpler denotational semantics, cf. Kammar *et al.*.

The second modification is our main contribution. We incorporate Hindley-Milner polymorphism in a standard way, without any value restriction. We indicate these latter modifications by *shading* in the figures. Amongst these:

- Local effect signatures Σ are finite mappings from operations op to pairs of value types A, B, whose action we denote by $(\mathsf{op} : A \to B) \in \Sigma$. We denote the restriction of a signature Σ to the set of operations disjoint from a given set $\Delta = \{\mathsf{op}_i \mid 1 \le i \le n\}$ by $\Sigma \setminus \Delta$.
- We extend the value types with *type variables* α and add *type variable environments* Θ, which are just finite sets of type variables.
- We introduce *schemes* $\forall \vec{\alpha}.A$, where $\vec{\alpha}$ denotes a finite set of $|\vec{\alpha}|$-many type variables ranged over by α_i.
- We introduce *kinding judgements* $\Theta \vdash X$ to explicitly keep track of the free type variables in X. The shorthand $\Theta \vdash X_1, \ldots, X_n$ stands for the conjunction of the judgements $\Theta \vdash X_1, \ldots, \Theta \vdash X_n$.
- Typing judgements $\Theta; \Xi; \Gamma \vdash M : X$ include the standard monomorphic environments Γ which are a unique assignment of types to variables. We extend those with type variable environments Θ and *polymorphic environments* Ξ, which are a unique assignment of schemes to variables. We assume that no variable can appear in both Γ and Ξ.[2] These polymorphic variables can be specialised at any type.
- We add *scheme judgements* whose effect annotation is outside the scope of the quantifier. The kinding assumption $\Theta \vdash \Sigma$ ensures that none of the type variables $\vec{\alpha}$

[2] This separation into two environments is not strictly necessary, as a monomorphic environment Γ may be identified with a polymorphic environment where each quantifier ranges over an empty tuple of type variables. We choose to separate the two to highlight which parts of the language interact with polymorphism.

Well-formed value types:

$$\frac{\alpha \in \Theta}{\Theta \vdash \alpha} \qquad \frac{}{\Theta \vdash \mathsf{bool}} \qquad \frac{\Theta \vdash A \qquad \Theta \vdash \underline{C}}{\Theta \vdash A \to \underline{C}} \qquad \frac{\Theta \vdash \underline{C} \qquad \Theta \vdash \underline{D}}{\Theta \vdash \underline{C} \Rightarrow \underline{D}}$$

Well-formed computation types, schemes, and effect signatures:

$$\frac{\Theta \vdash A \qquad \Theta \vdash \Sigma}{\Theta \vdash A\,!\,\Sigma} \qquad \frac{\Theta, \vec{\alpha} \vdash A}{\Theta \vdash \forall \vec{\alpha}.A} \qquad \frac{[\Theta \vdash A_i \qquad \Theta \vdash B_i]_{1 \le i \le n}}{\Theta \vdash \{\mathsf{op}_1 : A_1 \to B_1, \ldots, \mathsf{op}_n : A_n \to B_n\}}$$

$$\frac{[\Theta \vdash A]_{(x:A) \in \Gamma}}{\Theta \vdash \Gamma} \qquad \frac{[\Theta \vdash \forall \vec{\alpha}.A]_{(x:\forall \vec{\alpha}.A) \in \Xi}}{\Theta \vdash \Xi}$$

Value judgements $\boxed{\Theta;\Xi;\Gamma \vdash v : A}$***, assuming*** $\Theta \vdash \Xi, \Gamma, A$***:***

$$\frac{(x : A) \in \Gamma}{\Theta;\Xi;\Gamma \vdash x : A} \qquad \frac{(x : \forall \vec{\alpha}.B) \in \Xi \qquad [\Theta \vdash A_i]_{1 \le i \le |\vec{\alpha}|}}{\Theta;\Xi;\Gamma \vdash x : B[A_i/\alpha_i]_{1 \le i \le |\vec{\alpha}|}} \qquad \frac{}{\Theta;\Xi;\Gamma \vdash \mathbf{true} : \mathsf{bool}}$$

$$\frac{}{\Theta;\Xi;\Gamma \vdash \mathbf{false} : \mathsf{bool}} \qquad \frac{\Theta;\Xi;\Gamma, x : A \vdash c : \underline{C}}{\Theta;\Xi;\Gamma \vdash \mathbf{fun}\; x \mapsto c : A \to \underline{C}}$$

$$\frac{\begin{array}{c}\Theta;\Xi;\Gamma, x : A \vdash c_r : B\,!\,\Sigma' \qquad \Sigma \setminus \{\mathsf{op}_i \mid 1 \le i \le n\} \subseteq \Sigma' \\ \left[(\mathsf{op}_i : A_i \to B_i) \in \Sigma \qquad \Theta;\Xi;\Gamma, x_i : A_i, k_i : B_i \to B\,!\,\Sigma' \vdash c_i : B\,!\,\Sigma'\right]_{1 \le i \le n}\end{array}}{\Theta;\Xi;\Gamma \vdash \mathbf{handler}\;\{\mathbf{return}\; x \mapsto c_r, \mathsf{op}_1(x_1;k_1) \mapsto c_1, \ldots, \mathsf{op}_n(x_n;k_n) \mapsto c_n\} : A\,!\,\Sigma \Rightarrow B\,!\,\Sigma'}$$

Computation judgements $\boxed{\Theta;\Xi;\Gamma \vdash c : A\,!\,\Sigma}$***, assuming*** $\Theta \vdash \Xi, \Gamma, A$***:***

$$\frac{\Theta;\Xi;\Gamma \vdash v : A}{\Theta;\Xi;\Gamma \vdash \mathbf{return}\; v : A\,!\,\Sigma} \qquad \frac{\Theta;\Xi;\Gamma \vdash c_1 : (\forall \vec{\alpha}.A)\,!\,\Sigma \qquad \Theta;\Xi, x : \forall \vec{\alpha}.A;\Gamma \vdash c_2 : B\,!\,\Sigma}{\Theta;\Xi;\Gamma \vdash \mathbf{do}\; x \leftarrow c_1 \;\mathbf{in}\; c_2 : B\,!\,\Sigma}$$

$$\frac{(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}) \in \Sigma \qquad \Theta;\Xi;\Gamma \vdash v : A_{\mathsf{op}} \qquad \Theta;\Xi;\Gamma, y : B_{\mathsf{op}} \vdash c : A\,!\,\Sigma}{\Theta;\Xi;\Gamma \vdash \mathsf{op}(v; y.c) : A\,!\,\Sigma}$$

$$\frac{\Theta;\Xi;\Gamma \vdash v : \mathsf{bool} \qquad \Theta;\Xi;\Gamma \vdash c_1 : \underline{C} \qquad \Theta;\Xi;\Gamma \vdash c_2 : \underline{C}}{\Theta;\Xi;\Gamma \vdash \mathbf{if}\; v \;\mathbf{then}\; c_1 \;\mathbf{else}\; c_2 : \underline{C}}$$

$$\frac{\Theta;\Xi;\Gamma \vdash v_1 : A \to \underline{C} \qquad \Theta;\Xi;\Gamma \vdash v_2 : A}{\Theta;\Xi;\Gamma \vdash v_1\, v_2 : \underline{C}} \qquad \frac{\Theta;\Xi;\Gamma \vdash v : \underline{C} \Rightarrow \underline{D} \qquad \Theta;\Xi;\Gamma \vdash c : \underline{C}}{\Theta;\Xi;\Gamma \vdash \mathbf{with}\; v \;\mathbf{handle}\; c : \underline{D}}$$

Scheme judgement $\boxed{\Theta;\Xi;\Gamma \vdash c : (\forall \vec{\alpha}.A)\,!\,\Sigma}$***, assuming*** $\Theta \vdash \Xi, \Gamma, (\forall \vec{\alpha}.A), \Sigma$***:***

$$\frac{\Theta, \vec{\alpha};\Xi;\Gamma \vdash c : A\,!\,\Sigma}{\Theta;\Xi;\Gamma \vdash c : (\forall \vec{\alpha}.A)\,!\,\Sigma}\,(\textsc{Gen})$$

Fig. 4. A polymorphic type-and-effect system for effect handlers

appears in Σ. It is at this point that the hypothesised type inference algorithm should decide which type variables $\vec{\alpha}$ will be generalised. Our choice to separate scheme judgements from type judgements simplifies the let-rule, and makes it very similar to its standard, monomorphic counterpart.

The remaining kinding and typing rules are standard. Fine-grained call-by-value functions take values and perform computations. An operation invocation is well-typed if the type assigned to it by the local signature agrees with the type of the given parameter value v, and with the type of argument the continuation c expects. A handler is well-typed if the type of result the return clause expects matches with the type of computation the handler can handle, and each operation clause is well-typed when the parameter type and continuation type match the local signature the handler can handle. All clauses may cause additional effects, and their effect annotations must agree and include these operations, as well as any effect operations the handler does not explicitly handle, reflecting the fact that our handlers are forwarding. The fact that our handlers are deep is reflected by the type of the continuation: the effects the continuation may cause have already been handled, and so the continuation may cause effects in the resulting signature and of the resulting return type.

For the given effect system, we then have:

Theorem (Safety). *If $\vdash c : A \,!\, \Sigma$ holds, then either:*

(i) $c \rightsquigarrow c'$ *for some* $\vdash c' : A \,!\, \Sigma$*;*
(ii) $c = \mathbf{return}\ v$ *for some* $\vdash v : A$*; or*
(iii) $c = \mathsf{op}(v; y.\, c')$ *for some* $(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}) \in \Sigma$*,* $\vdash v : A_{\mathsf{op}}$*, and* $y : B_{\mathsf{op}} \vdash c' : A \,!\, \Sigma$*.*

In particular, when $\Sigma = \varnothing$, evaluation will not get stuck before returning a value. For a calculus that differs from ours only in being set in a call-by-push-value (Levy, 2004) rather than fine-grain call-by-value setting, Kammar *et al.* (2013) strengthen the result and show that all well-typed programs terminate. Such a result also holds in this case with a standard proof. We do not pursue such a proof here as it is orthogonal to our goal.

Proof

We prove progress and preservation lemmata separately by induction. We formalized[1] the calculus and the safety theorem in the Twelf proof assistant (Pfenning & Schürmann, 1999). Our formalization extends Bauer & Pretnar's (2014) existing formalization of Eff's core calculus with type schemes and polymorphism. The code is compatible with version 1.7.1 of Twelf. We summarise the crucial step, namely proving type-and-effect preservation under the DO-OP transition.

Assume that the reduct in DO-OP is well-typed, and invert its type derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}) \in \Sigma} \quad \dfrac{\vdots}{\Theta, \vec{\alpha} \vdash v : A_{\mathsf{op}}} \quad \dfrac{\vdots}{\Theta, \vec{\alpha}; y : B_{\mathsf{op}} \vdash c_1 : A \,!\, \Sigma}}{\Theta, \vec{\alpha} \vdash \mathsf{op}(v; y.\, c_1) : A!\Sigma}}{\Theta \vdash \mathsf{op}(v; y.\, c_1) : (\forall \vec{\alpha}.A)!\Sigma} \quad \dfrac{\vdots}{\Theta; x : \forall \vec{\alpha}.A \vdash c_2 : B \,!\, \Sigma}}{\Theta \vdash \mathbf{do}\, x \leftarrow \mathsf{op}(v; y.\, c_1)\ \mathbf{in}\ c_2 : B \,!\, \Sigma}$$

The GEN rule ensures that none of the type variables in $\vec{\alpha}$ appear in Σ. Because Σ includes $\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}$, none of these variables appear in A_{op}, and we may strengthen the derivation of $\Theta, \vec{\alpha} \vdash v : A_{\mathsf{op}}$ to a derivation of $\Theta \vdash v : A_{\mathsf{op}}$. As a consequence, the following derivation is valid:

$$\dfrac{\dfrac{}{(\mathsf{op} : A_{\mathsf{op}} \to B_{\mathsf{op}}) \in \Sigma} \quad \dfrac{\vdots}{\Theta \vdash v : A_{\mathsf{op}}} \quad \dfrac{\dfrac{\dfrac{\vdots}{\Theta, \vec{\alpha}; y : B_{\mathsf{op}} \vdash c_1 : A \,!\, \Sigma}}{\Theta; y : B_{\mathsf{op}} \vdash c_1 : (\forall \vec{\alpha}. A) \,!\, \Sigma} \quad \dfrac{\vdots}{\Theta; x : \forall \vec{\alpha}. A \vdash c_2 : B \,!\, \Sigma}}{\Theta; y : B_{\mathsf{op}} \vdash \mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2 : B \,!\, \Sigma}}{\Theta \vdash \mathsf{op}(v; y.\, \mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2) : B \,!\, \Sigma}$$

Therefore, the reduction in DO-OP preserves both the type and the effect annotation. ■

The Safety Theorem is robust under the following standard variations in the calculus:

coarse annotations. We can make the signature Σ global, and only keep track of which operations are used, as in Pretnar (2015). The types in this global signature cannot use any type variables. The soundness proof remains essentially unchanged[1]. Due to the lack of type variables in the global signature, there is no need to impose a side-condition on the well-formedness of the effect annotation in the GEN rule.

It may seem this coarser system is a restriction of our current system, where the type information for each operation has to agree in all effect annotations, and hence it is sound by the Safety Theorem. This is not the case. In this coarser system, the signatures on function types are not annotated with the types of the operations. If those types were fully written out, they would involve the global signature, leading to potential mutual recursion between signatures and function types. For example, if we elaborate the global signature $\Sigma = \{\mathsf{op} : \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit} \,!\{\mathsf{op}\})\}$, we would get:

$$\Sigma = \{\mathsf{op} : \mathsf{unit} \to (\mathsf{unit} \to (\mathsf{unit} \,!\, \Sigma))\}$$

where the outermost arrow is part of the signature syntax and receives no effect annotation on the co-domain. This recursion is not a mere formality. As mentioned above, the type-and-effect system with local signatures we have described ensures well-typed terms terminate, cf. Kammar *et al.* (2013). When we switch to a global signature, we can use effect operations with higher-order return types to express well-typed diverging computations. With the above global signature $\Sigma = \{\mathsf{op} : \mathsf{unit} \to (\mathsf{unit} \to \mathsf{unit} \,!\{\mathsf{op}\})\}$, consider the handler

$$\begin{aligned} H := \mathbf{handler}\, \{ & \mathbf{return}\, x \mapsto \mathbf{return}\, x, \\ & \mathsf{op}(\_; k) \mapsto k(\mathbf{fun}\, \_ \mapsto \mathsf{op}\,()\,())\} \end{aligned}$$

In the coarse type system, we can derive the judgement:

$$\vdash H : (\mathsf{unit} \,!\{\mathsf{op}\}) \Rightarrow (\mathsf{unit} \,!\, \varnothing)$$

Handling the computation $\vdash \mathsf{op}\,()\,() : \mathsf{unit} \,!\{\mathsf{op}\}$ with H diverges:

$$\begin{aligned} \mathbf{with}\, H\, \mathbf{handle}\, \mathsf{op}()\,() & \leadsto^{+} \mathbf{with}\, H\, \mathbf{handle}\, (\mathbf{fun}\, \_ \mapsto \mathsf{op}()\,())\,() \\ & \leadsto \mathbf{with}\, H\, \mathbf{handle}\, \mathsf{op}()\,() \end{aligned}$$

In fact, by a variation on Landin's (1964) knot, we can express a variant of the Y-combinator, such that for a function f that is pure, $Y\,f$ behaves like the fixed-point of f when invoked on pure arguments.

no annotations. We can remove all the effect annotations Σ from type judgements and fix a single, global signature Σ. The advantage of having an effect system is the additional guarantee in clause *(iii)* of the Safety Theorem, which ensures that any unhandled operation must appear in Σ. Without annotations, any operation may be called. This system is a restriction of the coarse variation, where each effect annotation is the entire signature. Consequently, it is sound.

additional language features. To the calculus with coarse annotations, we can add fixed-points, *structural subtyping* and static *effect instances*. The soundness proof remains essentially unchanged[1] as these modifications are orthogonal to polymorphism. Similarly, we can replace deep handlers with shallow ones[1], as in Kammar *et al.* (2013) and Kiselyov *et al.* (2013). As the changes are again orthogonal to polymorphism, we may reasonably assume a similar soundness result to hold for a calculus that incorporates all of the above: subtyping, instances for coarse annotations, and, through two separate syntactic constructs, both deep and shallow handlers.

While the strong normalisation property of the fully annotated calculus shows non-termination cannot be admitted through the back-door, it does not mean recursion cannot be safely integrated with polymorphism through an explicit fixed-point construct. Compare the situation with, for example, the simply-typed λ-calculus and its sound extension with a fixed-point operator in the PCF calculus (Scott, 1993; Plotkin, 1977). We conjecture it is straightforward to add an appropriate fixed-point operator to our fully annotated calculus safely, and the fact that the coarsely annotated calculus can be safely extended with fixed-points supports this conjecture.

4 Expressiveness

There is currently no simple type system integrating reference cells with polymorphism without the value restriction. This non-existence contrasts the simplicity of our type system, and calls into question both its degree of feature integration and its expressiveness. First, we evaluate the degree and smoothness of the interaction between polymorphism and other features in our calculus. Then, we highlight the difference in expressiveness between effect handlers and reference cells. As a basis for our evaluation and comparison, we use Leroy's (1992) set of example programs for analysing the usefulness of a polymorphic type system for reference cells.

4.1 Evaluation

Algebraic effects allow us to lace a piece of code with operations in the signature

$$\{\mathsf{get} : \mathsf{unit} \to \alpha, \mathsf{set} : \alpha \to \mathsf{unit}\}$$

The scheme assigned to the handler H_{ST}, which handles them away, is

$$H_{ST} : \forall \alpha, \beta . \alpha \,!\{\mathsf{get} : \mathsf{unit} \to \beta, \mathsf{set} : \beta \to \mathsf{unit}\} \Rightarrow (\beta \to \alpha \,!\, \varnothing) \,!\, \varnothing$$

It takes a computation of type α that interacts with a state of type β, and handles it to a pure function of type $\beta \to \alpha \,!\, \varnothing$. The rightmost $\varnothing$ indicates that no effects can occur when producing the function.

This handler can handle computations with different types of state, for example:

$$(\textbf{with}\ H_{ST}\ \textbf{handle}\ \mathsf{set}())\ ();$$
$$(\textbf{with}\ H_{ST}\ \textbf{handle}\ \mathsf{get}())\ \textbf{true}$$

We can also use effects in polymorphic code:

```
do $f \leftarrow$ if $\mathsf{get}()$ then return fun $x\,y \mapsto$ return $x$   $(*\ f : \forall\alpha.\alpha \to (\alpha \to \alpha\,!\,\varnothing)\,!\,\varnothing\ *)$
                     else return fun $x\,y \mapsto$ return $y$
 in $(f$ (fun $b \mapsto$ return $b)$                              $(*\ \alpha := \mathsf{bool} \to \mathsf{bool}\,!\{\mathsf{get}\}\ *)$
        (fun $b \mapsto \mathsf{set}(b)$; return $b))$
      $(f$ true false$)$                                            $(*\ \alpha := \mathsf{bool}\ *)$
```

In our call-by-value semantics, if we wrap this computation with the state handler, the memory look-up in f's definition will only occur once.

To demonstrate that the polymorphic, effectful, and high-order features interact well, we hypothetically extend our calculus with lists. The hypothesised extension may include primitives such as the empty list [ ], a list cons $(::)$ and tail-recursive iteration **foldl**, which we expect to interact smoothly with polymorphism. Thus we can use H_{ST} to implement functional features in an imperative style.

```
do imp_map $\leftarrow$ fun $f\,xs \mapsto$
   with $H_{ST}$ handle (foldl  (fun $x \mapsto \mathsf{set}\,(f\ x :: \mathsf{get}())$)
                                $()$
                                $xs$;
                         $\mathsf{reverse}\,(\mathsf{get}())$
   [ ] (* initial state *) in ...
```

The scheme assigned to imp_map is

$$\text{imp\_map} : \forall\alpha\beta.(\alpha \to \beta\,!\,\Sigma) \to (\alpha\ \mathsf{list} \to \beta\ \mathsf{list}\,!\,\Sigma)\,!\,\varnothing$$

for any Σ. This implementation is imperative in style, but not imperative *per se*, as all operations are handled by high-order functions. The function imp_map can also be partially applied and retain its polymorphism, for example, in

```
do list_id $\leftarrow$ imp_map id in
do nil     $\leftarrow$ list_id [ ]   in ...
```

we have the scheme assignments:

$$\begin{aligned}\text{list\_id} &: \forall\alpha.\alpha\ \mathsf{list} \to \alpha\ \mathsf{list}\,!\,\varnothing\\ \text{nil} &: \forall\alpha.\alpha\ \mathsf{list}\end{aligned}$$

Most importantly, the following program is well-typed:

$$\textbf{do}\ id \leftarrow (\textbf{fun}\ f \mapsto f)\ (\textbf{fun}\ x \mapsto x)\ \textbf{in}$$
$$\textbf{do}\ id' \leftarrow id\,(id)\ \textbf{in}\ \ldots$$

and both functions are assigned the polymorphic scheme $\forall\alpha.\alpha \to \alpha!\Sigma$. Such mixed-variance polymorphism is ruled out by all current variants of the value restriction.

4.2 Reference cells

We conjecture it is impossible to simulate full blown reference cells using effect handlers without other language features, but we do not have a formal proof for this statement. We can increase modularity by introducing instances (Bauer & Pretnar, 2015, 2014; Pretnar, 2014). These may be thought of as first class atomic names. With instances, each effect instance ι and an operation symbol op determine an operation $\iota\#\mathsf{op}$. In handlers, each operation clause $v\#\mathsf{op}(x;k) \mapsto c$ specifies which instance, dynamically given by the value v, of the statically chosen effect operation symbol op the handler handles. At runtime, invocations of the same operation op but with different instances will not be caught by this handler and will be forwarded.

Instances allow us to pass a cell around by passing an instance, but they are still less expressive than having the ability to allocate arbitrarily many new cells dynamically. For example, we do not know how to implement even the simplest of Leroy's (1992) benchmarks:

$$\mathbf{do}\ \mathrm{make\_ref} \leftarrow \mathbf{fun}\ x \mapsto\ \mathbf{ref}\ \ x\ \mathbf{in}\ \ldots$$

Eff provides a mechanism that can both generate *fresh* instances and attach them to a stateful *resource* (Bauer & Pretnar, 2015), allowing one to directly implement a make_ref analogue: make_ref creates a fresh instances that has get and set operations associated with it. Only code that knows what the instance is, can handle these effects. However, it is not easy to find a corresponding type-and-effect system for fresh instances (Bauer & Pretnar, 2014; Pretnar, 2014), let alone a polymorphic one. Without an alternative to reference cells, the expressiveness of our calculus is limited.

As a final example, recall the problematic reference cell example which cannot be directly expressed in our calculus:

$$\begin{aligned}&\mathbf{do}\ r \leftarrow\ \mathbf{ref}\ \ [\,]\ \mathbf{in}\\ &r := [()];\\ &\mathbf{true} :: !r\end{aligned}$$

We can express a computation that writes a $\mathsf{unit\ list}$ value and reads a $\mathsf{bool\ list}$ value:

$$\begin{aligned}&\mathsf{set}([()]);\\ &\mathbf{true} :: \mathsf{get}()\end{aligned}$$

However, this computation has the effect annotation

$$\{\mathsf{set} : \mathsf{unit\ list} \to \mathsf{unit}, \mathsf{get} : \mathsf{unit} \to \mathsf{bool\ list}\}$$

which is incompatible with the type of the state handler H_{ST}. Other handlers for the state operations may have a compatible type. For example, the read-only state handler H_{RO} which ignores any memory updates:

$$\begin{aligned}H_{RO} := \mathbf{handler}\ \{\ &\mathbf{return}\ x \mapsto \mathbf{fun}\ \_ \mapsto \mathbf{return}\ x,\\ &\mathsf{get}(\_;k) \mapsto \mathbf{fun}\ s \mapsto k\ s\ s,\\ &\mathsf{set}(\_;k) \mapsto \mathbf{fun}\ s \mapsto k\ ()\ s\}\end{aligned}$$

Syntax

p	$::=$	$\mathtt{p} \mid \mathtt{q} \mid \mathtt{r} \mid \ldots$	parameter
v	$::=$		value
		x	variable
	$\mid$	**true** $\mid$ **false**	boolean constants
	$\mid$	$()$	unit value
	$\mid$	**fun** $x \mapsto c$	function
c	$::=$		computation
		return v	return
	$\mid$	**do** $x \leftarrow c_1$ **in** c_2	sequencing
	$\mid$	**if** v **then** c_1 **else** c_2	conditional
	$\mid$	$v_1\, v_2$	application
	$\mid$	$!p$	dereferencing
	$\mid$	$p := v$	assignment
	$\mid$	**dlet** $p \leftarrow v$ **in** c	rebinding

Fig. 5. A calculus for dynamically scoped state

It has the scheme

$$H_{RO} : \forall \alpha, \beta, \gamma. \alpha \,!\{\mathsf{get} : \mathsf{unit} \to \beta, \mathsf{set} : \gamma \to \mathsf{unit}\} \Rightarrow (\beta \to \alpha \,!\, \varnothing) \,!\, \varnothing$$

and can be applied to the above computation without run-time errors.

4.3 Dynamically scoped state

As we saw in Sec. 2.2, we can simulate global state using the handler H_{ST}, and handle this state locally to give a pure computation. While we do not know whether effect handlers can simulate reference cells or not, we will now characterise the handler H_{ST} as expressing the notion of *dynamically scoped* state.

In order to explain what we mean by dynamically scoped state, and to make the discussion precise, we consider the calculus presented in Fig. 5. It is a fine-grained call-by-value variation on the dynamic scope calculi of Kiselyov *et al.* (2006) and Moreau (1998).

We assume a set of parameters ranged over by p that name dynamically scoped memory cells. These cells can be dereferenced, $!p$, or assigned to, $p := v$, just like ref cells. The rebinding construct **dlet** $p \leftarrow v$ **in** c declares that in executing c, all references to p will be bound to this occurrence of p, and shadow other binding declarations that may be in place.

Auxiliary definitions

Evaluation contexts:

$$E \quad ::= \quad [\,] \mid E[\mathbf{do}\, x \leftarrow [\,]\, \mathbf{in}\, c] \mid E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, [\,]]$$

Parameter binding:

$$\mathrm{bp}([\,]) := \varnothing \qquad \mathrm{bp}(E[\mathbf{do}\, x \leftarrow [\,]\, \mathbf{in}\, c]) := \mathrm{bp}(E) \qquad \mathrm{bp}(E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, [\,]]) := \mathrm{bp}(E) \cup \{p\}$$

Semantics

$$E[\mathbf{do}\, x \leftarrow \mathbf{return}\, v\, \mathbf{in}\, c] \overset{dyn}{\rightsquigarrow} E[c[v/x]] \qquad E[\mathbf{if\ true\ then}\, c_1\, \mathbf{else}\, c_2] \overset{dyn}{\rightsquigarrow} E[c_1]$$

$$E[\mathbf{if\ false\ then}\, c_1\, \mathbf{else}\, c_2] \overset{dyn}{\rightsquigarrow} E[c_2] \qquad E[(\mathbf{fun}\, x \mapsto c)\, v] \overset{dyn}{\rightsquigarrow} E[c[v/x]]$$

$$E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in\ return}\, v'] \overset{dyn}{\rightsquigarrow} E[\mathbf{return}\, v']$$

$$\frac{}{E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, E'[!p]] \overset{dyn}{\rightsquigarrow} E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, E'[\mathbf{return}\, v]]}\,(p \notin \mathrm{bp}(E'))$$

$$\frac{}{E[\mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, E'[p := v']] \overset{dyn}{\rightsquigarrow} E[\mathbf{dlet}\, p \leftarrow v'\, \mathbf{in}\, E'[\mathbf{return}\, ()]]}\,(p \notin \mathrm{bp}(E'))$$

Fig. 6. The semantics for dynamically scoped state

For example, assuming we have a type of integers the following code will evaluate to **return** 2.

```
do f ← dlet p ← 0 in
           return (fun _ ↦
                       p := 1+!p) in
  dlet p ← 1 in
     f ();
     !p
```

During its execution, the state changes inside the function f bind dynamically to the closest enclosing rebinding, which is the second one.

Fig. 6 describes the (Felleisen-style) operational semantics for this calculus. We kept the style of semantics as close as possible to Kiselyov *et al.*'s (2006) to make it clear we use the same notion of dynamic scope, and our theoretical treatment closely mirrors theirs. The semantics uses the set of parameters bound in a given context E, denoted by $\mathrm{bp}(E)$. The three shaded transitions are the transitions specific to dynamic scope. First, a fully evaluated computation removes a preceding parameter binding, as it will no longer be used. For the other two transitions, the side condition $p \notin \mathrm{bp}(E')$ ensures the uniqueness of the decomposition into the context E' by locating the closest rebinding of p. The semantics of dereferencing returns the value associated to this closest rebinding, while the semantics of assignment modifies it. In our design, assignment evaluates to the unit value, deviating

Term-level translation

$$\lceil x \rceil := x \qquad \lceil \textbf{true} \rceil := \textbf{true} \qquad \lceil \textbf{false} \rceil := \textbf{false} \qquad \lceil \textbf{fun}\, x \mapsto c \rceil := \textbf{fun}\, x \mapsto \lceil c \rceil$$

$$\lceil v_1\ v_2 \rceil := \lceil v_1 \rceil\, \lceil v_2 \rceil \qquad \lceil \textbf{return}\, v \rceil := \textbf{return}\, \lceil v \rceil \qquad \lceil \textbf{do}\, x \leftarrow c_1\, \textbf{in}\, c_2 \rceil := \textbf{do}\, x \leftarrow \lceil c_1 \rceil\, \textbf{in}\, \lceil c_2 \rceil$$

$$\lceil !p \rceil := \texttt{get\_}p() \qquad \lceil p := v \rceil := \texttt{set\_}p(\lceil v \rceil) \qquad \lceil \textbf{dlet}\, p \leftarrow v\, \textbf{in}\, c \rceil := (\textbf{with}\, H^p_{ST}\, \textbf{handle}\, \lceil c \rceil)\, \lceil v \rceil$$

where:

$$\begin{aligned} H^p_{ST} := \textbf{handler}\ \{ & \texttt{get\_}p(\_;k) \mapsto \textbf{return}\,(\textbf{fun}\, s \mapsto (k\ s)\ s), \\ & \texttt{set\_}p(s';k) \mapsto \textbf{return}\,(\textbf{fun}\, \_ \mapsto (k\ ())\ s'), \\ & \textbf{return}\, x \quad\ \mapsto \textbf{return}\,(\textbf{fun}\, \_ \mapsto \textbf{return}\, x)\} \end{aligned}$$

Fig. 7. Handlers expressing dynamically scoped state

from Kiselyov *et al.*'s semantics. This purely cosmetic change does not alter the nature of dynamically scope state we are dealing with, and makes the relationship with H_{ST} tighter.

The example above evaluates as follows:

$$\begin{array}{l} \textbf{do}\, f \leftarrow \textbf{dlet}\, \texttt{p} \leftarrow 0\, \textbf{in} \\ \qquad\qquad \textbf{return}\,(\textbf{fun}\, \_ \mapsto \\ \qquad\qquad\qquad\qquad \texttt{p := 1+!p})\, \textbf{in} \\ \quad \textbf{dlet}\, \texttt{p} \leftarrow 1\, \textbf{in} \\ \qquad f\,(); \\ \qquad \texttt{!p} \end{array} \overset{dyn}{\rightsquigarrow} \begin{array}{l} \textbf{do}\, f \leftarrow \textbf{return}\,(\textbf{fun}\, \_ \mapsto \\ \qquad\qquad\qquad\qquad \texttt{p := 1+!p})\, \textbf{in} \\ \quad \textbf{dlet}\, \texttt{p} \leftarrow 1\, \textbf{in} \\ \qquad f\,(); \\ \qquad \texttt{!p} \end{array}$$

$$\overset{dyn}{\rightsquigarrow} \begin{array}{l} \textbf{dlet}\, \texttt{p} \leftarrow 1\, \textbf{in} \\ \quad (\textbf{fun}\, \_ \mapsto \\ \qquad \texttt{p := 1+!p})\,(); \\ \quad \texttt{!p} \end{array} \overset{dyn}{\rightsquigarrow} \begin{array}{l} \textbf{dlet}\, \texttt{p} \leftarrow 1\, \textbf{in} \\ \quad \texttt{p := 1+!p}; \\ \quad \texttt{!p} \end{array} \overset{dyn}{\rightsquigarrow}{}^{+} \quad \textbf{return}\, 2$$

Fig. 7 shows how effect handlers express dynamically scoped state. Using Felleisen's (1991) terminology, it is a *macro* translation. First, it does not use any information collected globally as it is defined homomorphically over the syntax of the language. Second, it keeps the common core of the two languages unchanged, translating a boolean value to itself, a function to a function, and so forth. The translation is straightforward: it translates dereferencing and assignments to p as specially named effects, $\texttt{get\_}p$ and $\texttt{set\_}p$. Rebinding amounts to handling with H_{ST}, and passing the translated rebinding value as the initial value.

This translation simulates dynamic allocation:

Theorem (Simulation)**.** *For all* $c \overset{dyn}{\rightsquigarrow} c'$, *we have* $\lceil c \rceil \rightsquigarrow^{+} \lceil c' \rceil$.

Proof

First, extend the translations to evaluation contexts, and show that $\lceil E[c] \rceil = \lceil E \rceil[\lceil c \rceil]$. Then, show the translation respects capture-avoiding substitution: $\lceil c[v/x] \rceil = \lceil c \rceil[\lceil v \rceil/x]$. To deal with the mismatch between Felleisen-style and small-step semantics, show that for

Types

A,B	$::=$		value type
		α	type variable
	$\mid$	bool	boolean type
	$\mid$	unit	unit type
	$\mid$	$A \to B$	function type
		$\forall\vec{\alpha}.A$	scheme
Σ	$::=$	$\{p_1 : A_1, \ldots, p_n : A_n\}$	parameter signature
Θ	$::=$	$\{\alpha_1, \ldots, \alpha_n\}$	type variable environment
Γ	$::=$	$\varnothing \mid \Gamma, x : A$	monomorphic environment
Ξ	$::=$	$\varnothing \mid \Xi, x : \forall\vec{\alpha}.A$	polymorphic environment

Fig. 8. Polymorphic types for dynamically scoped state

all evaluation contexts E, if $c \overset{dyn}{\rightsquigarrow} c'$ then $\lceil E \rceil[c] \leadsto^{+} \lceil E \rceil[c']$. It therefore suffices to prove the theorem for each of the transitions in Fig. 6 specialised to $E := [\ ]$.

For each of the common constructs of the two calculi, the proof is immediate, for example:

$$\lceil \mathbf{do}\, x \leftarrow \mathbf{return}\, v\, \mathbf{in}\, c \rceil = \mathbf{do}\, x \leftarrow \mathbf{return}\, \lceil v \rceil\, \mathbf{in}\, \lceil c \rceil \leadsto \lceil c \rceil[\lceil v \rceil/x] = \lceil c[v/x] \rceil$$

The next remaining transition amounts to handling a terminal computation:

$$\begin{aligned}\lceil \mathbf{dlet}\, \mathtt{p} \leftarrow v\, \mathbf{in}\, \mathbf{return}\, v' \rceil &= (\mathbf{with}\, H^{p}_{ST}\, \mathbf{handle}\, \mathbf{return}\, \lceil v' \rceil)\ \lceil v \rceil \\ &\leadsto^{+} (\mathbf{fun}\, \_ \mapsto \mathbf{return}\, \lceil v' \rceil)\ \lceil v \rceil \leadsto \mathbf{return}\, \lceil v' \rceil\end{aligned}$$

For the final two transitions, show that, for all contexts E, parameters $p \notin \mathrm{bp}(E)$, operations op that is either $\mathtt{get}\_p$ or $\mathtt{set}\_p$, and x fresh for E, we have:

$$\lceil E \rceil[\mathsf{op}(v; x.\, c)] \leadsto^{*} \mathsf{op}(v; x.\, \lceil E \rceil[c])$$

And finally, calculate:

$$\begin{aligned}\lceil \mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, E[!p] \rceil &= (\mathbf{with}\, H^{p}_{ST}\, \mathbf{handle}\, \lceil E \rceil[\mathtt{get}\_p((); x.\, \mathbf{return}\, x))\ \lceil v \rceil \\ &\leadsto^{*} (\mathbf{with}\, H^{p}_{ST}\, \mathbf{handle}\, \mathtt{get}\_p((); x.\, \lceil E \rceil[\mathbf{return}\, x]))\ \lceil v \rceil \\ &\leadsto^{+} (\mathbf{fun}\, s \mapsto ((\mathbf{fun}\, x \mapsto \mathbf{with}\, H^{p}_{ST}\, \mathbf{handle}\, \lceil E \rceil[\mathbf{return}\, x])\ s)\ s)\ \lceil v \rceil \\ &\leadsto^{+} \mathbf{with}\, H^{p}_{ST}\, \mathbf{handle}\, \lceil E \rceil[\mathbf{return}\, v]\ \lceil v \rceil \\ &= \lceil \mathbf{dlet}\, p \leftarrow v\, \mathbf{in}\, E[\mathbf{return}\, v] \rceil\end{aligned}$$

A similar calculation for assignment completes the proof. ■

This translation, while being straightforward, also preserves the type system. Fig. 8 presents the types for the calculus. The only notable feature is that, like Kiselyov *et al.*, we assume a global signature assigning to each parameter a type. As the signature is global, these (monomorphic) types do not contain any type variables.

Fig. 9 presents the kind and (Hindley-Milner polymorphic) type system for the calculus. The kind system ensures well-kinded signatures assign types without type variables.

Well-formed types, parameter signatures, and schemes:

$$\frac{\alpha \in \Theta}{\Theta \vdash^{\mathsf{dyn}} \alpha} \qquad \frac{}{\Theta \vdash^{\mathsf{dyn}} \mathsf{bool}} \qquad \frac{}{\Theta \vdash^{\mathsf{dyn}} \mathsf{unit}} \qquad \frac{\Theta \vdash^{\mathsf{dyn}} A \qquad \Theta \vdash^{\mathsf{dyn}} C}{\Theta \vdash^{\mathsf{dyn}} A \to C}$$

$$\frac{[\vdash^{\mathsf{dyn}} A_i]_{1 \le i \le n}}{\Theta \vdash^{\mathsf{dyn}} \{p_1 : A_1, \dots, p_n : A_n\}} \qquad \frac{\Theta, \vec{\alpha} \vdash^{\mathsf{dyn}} A}{\Theta \vdash^{\mathsf{dyn}} \forall \vec{\alpha}.A}$$

Well-formed polymorphic and monomorphic environments:

$$\frac{[\Theta \vdash^{\mathsf{dyn}} \forall \vec{\alpha}.A]_{(x:\forall \vec{\alpha}.A) \in \Xi}}{\Theta \vdash^{\mathsf{dyn}} \Xi} \qquad \frac{[\Theta \vdash^{\mathsf{dyn}} A]_{(x:A) \in \Gamma}}{\Theta \vdash^{\mathsf{dyn}} \Gamma}$$

Value judgements $\boxed{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v : A}$***, assuming*** $\Theta \vdash^{\mathsf{dyn}} \Xi, \Gamma, A, \Sigma$***:***

$$\frac{(x : A) \in \Gamma}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} x : A} \qquad \frac{(x : \forall \vec{\alpha}.B) \in \Xi \qquad [\Theta \vdash^{\mathsf{dyn}} A_i]_{1 \le i \le |\vec{\alpha}|}}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} x : B[A_i/\alpha_i]_{1 \le i \le |\vec{\alpha}|}} \qquad \frac{}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{true} : \mathsf{bool}}$$

$$\frac{}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{false} : \mathsf{bool}} \qquad \frac{}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} () : \mathsf{unit}} \qquad \frac{\Theta; \Xi; \Gamma, x : A \vdash^{\mathsf{dyn}}_{\Sigma} c : B}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{fun}\, x \mapsto c : A \to B}$$

Computation judgements $\boxed{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c : A}$***, assuming*** $\Theta \vdash^{\mathsf{dyn}}_{\Sigma} \Xi, \Gamma, A, \Sigma$***:***

$$\frac{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v : A}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{return}\, v : A} \qquad \frac{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c_1 : (\forall \vec{\alpha}.A) \qquad \Theta; \Xi, x : \forall \vec{\alpha}.A; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c_2 : B}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{do}\, x \leftarrow c_1 \,\mathbf{in}\, c_2 : B}$$

$$\frac{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v : \mathsf{bool} \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c_1 : C \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c_2 : C}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{if}\, v \,\mathbf{then}\, c_1 \,\mathbf{else}\, c_2 : C}$$

$$\frac{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v_1 : A \to B \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v_2 : A}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v_1\, v_2 : B} \qquad \frac{(p : A) \in \Sigma}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} !p : A}$$

$$\frac{(p : A) \in \Sigma \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v : A}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} p := v : \mathsf{unit}} \qquad \frac{(p : A) \in \Sigma \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} v : A \qquad \Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c : B}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} \mathbf{dlet}\, p \leftarrow v \,\mathbf{in}\, c : B}$$

Scheme judgement $\boxed{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c : (\forall \vec{\alpha}.A)}$***, assuming*** $\Theta \vdash^{\mathsf{dyn}}_{\Sigma} \Xi, \Gamma, (\forall \vec{\alpha}.A), \Sigma$***:***

$$\frac{\Theta, \vec{\alpha}; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c : A}{\Theta; \Xi; \Gamma \vdash^{\mathsf{dyn}}_{\Sigma} c : (\forall \vec{\alpha}.A)}\,(\textsc{gen})$$

Fig. 9. A polymorphic type system for dynamically scoped state

Type-level translation with effect annotations

$$\lceil \alpha \rceil := \alpha \qquad \lceil \mathsf{bool} \rceil := \mathsf{bool} \qquad \boxed{\lceil A \to B \rceil := \lceil A \rceil \to \lceil B \rceil ! \lceil \Sigma \rceil} \qquad \lceil \forall \vec{\alpha}.A \rceil := \forall \vec{\alpha}.\lceil A \rceil$$

$$\lceil \Theta \rceil := \Theta \qquad \lceil \Gamma \rceil := \{x : \lceil A \rceil \mid (x : A) \in \Gamma\} \qquad \lceil \Xi \rceil := \{x : \forall \vec{\alpha}.\lceil A \rceil \mid (x : \forall \vec{\alpha}.A) \in \Gamma\}$$

$$\boxed{\lceil \Sigma \rceil := \{\mathtt{get\_}p : \mathsf{unit} \to \lceil A \rceil, \mathtt{set\_}p : \lceil A \rceil \to \mathsf{unit} \mid (p : A) \in \Sigma\}}$$

provided Σ is ground.

Type-level translation without effect annotations

$$\lfloor \alpha \rfloor := \alpha \qquad \lfloor \mathsf{bool} \rfloor := \mathsf{bool} \qquad \boxed{\lfloor A \to B \rfloor := \lfloor A \rfloor \to \lfloor B \rfloor} \qquad \lfloor \forall \vec{\alpha}.A \rfloor := \forall \vec{\alpha}.\lfloor A \rfloor \qquad \lfloor \Theta \rfloor := \Theta$$

$$\lfloor \Gamma \rfloor := \{x : \lfloor A \rfloor \mid (x : A) \in \Gamma\} \qquad \lfloor \Xi \rfloor := \{x : \forall \vec{\alpha}.\lfloor A \rfloor \mid (x : \forall \vec{\alpha}.A) \in \Gamma\}$$

for the ambient effect signature:

$$\boxed{\lfloor \Sigma \rfloor := \{\mathtt{get\_}p : \mathsf{unit} \to \lfloor A \rfloor, \mathtt{set\_}p : \lfloor A \rfloor \to \mathsf{unit} \mid (p : A) \in \Sigma\}}$$

Fig. 10. Handlers type system expressing dynamically scoped state

Typing judgements $\Theta; \Xi; \Gamma \vdash^{\mathrm{dyn}}_{\Sigma} c : A$ refer to the fixed, ambient, well-kinded parameter signature Σ. The typing rules specific to dynamically scoped state (shaded) ensure that we may only dereference, assign to, and rebind a parameter in accordance with the ambient signature. The assignment rule also highlights our decision to ascribe the unit type to assignment, in a minor deviation from Kiselyov *et al.*. The (GEN) rule is now completely unrestricted, ensured by the assumption that the type signature does not involve type variables.

Fig. 10 extends the translation to types. The parameter signature Σ translates into an effect signature containing the distinct pair of effects corresponding to this parameter, namely `get_`p and `set_`p, with the appropriate type. Function types may cause any effect in this translated signature $\lceil \Sigma \rceil$. This translation is therefore not-well-defined: if Σ contains any function types, then $\lceil \Sigma \rceil$ refers to $\lceil A \to B \rceil$, which refers to $\lceil \Sigma \rceil$ again.

There are at least three ways around this issue. The simplest solution, presented in the top half of Fig. 10 is to restrict Σ to *ground* types, i.e., prohibit storing functions.

A less restrictive solution is to use the coarser type system for effect handlers that does not track effect annotations at all, and define $\lfloor A \to B \rfloor := \lfloor A \rfloor \to \lfloor B \rfloor$, as in the bottom half of Fig. 10. This solution works well, as the effects `get_`p and `put_`p maintain their type.

A more sophisticated potential solution is to use equi-recursive effect signatures. At this point in time, such a type-and-effect system has not been developed, but we do not foresee any serious obstacles in developing it: its denotational semantics would involve a recursive domain equation in the same spirit as in Bauer & Pretnar (2014).

The fact that higher-order parameter types merit domain-theoretic semantics is not surprising in light of a piece of folklore due to Oleg Kiselyov: such parameters allow non-terminating programs. We call a type A *inhabited* if there exists a closed value $\vdash^{\mathrm{dyn}}_{\Sigma} v : A$.

Proposition. *If Σ contains a higher-order type parameter $(p : A \to B) \in \Sigma$ for some inhabited type A, then there is a term c satisfying:*

$$c \overset{dyn+}{\rightsquigarrow} c$$

Proof
Let $\vdash^{\mathrm{dyn}}_{\Sigma} v : A$ be an inhabitant of A, and take:

$$\begin{aligned} c := \mathbf{dlet}\, p \leftarrow (\mathbf{fun}\, a \mapsto (!p)\, a)\ \mathbf{in} \\ (!p)\, v \end{aligned}$$

Then:

$$c \overset{dyn}{\rightsquigarrow} \begin{array}{c}\mathbf{dlet}\, p \leftarrow (\mathbf{fun}\, a \mapsto (!p)\, a)\ \mathbf{in} \\ (\mathbf{fun}\, a \mapsto (!p)\, a)\, v\end{array} \overset{dyn+}{\rightsquigarrow} \begin{array}{c}\mathbf{dlet}\, p \leftarrow (\mathbf{fun}\, a \mapsto (!p)\, a)\ \mathbf{in} \\ (!p)\, v\end{array} = c$$

as required. ■

Moreover, every parameter $(p : A \to B)$ lets us define a form of a fixed-point combinator $Y : ((A \to B) \to A \to B) \to (A \to B)$ by a variant of Landin's knot, provided the functions passed to this combinator and their arguments do not involve p.

The two proposed translations are correct:

Theorem (Type Preservation). *For every $\Theta;\Xi;\Gamma \vdash^{\mathrm{dyn}}_{\Sigma} c : A$ and $\Theta;\Xi;\Gamma \vdash^{\mathrm{dyn}}_{\Sigma} v : A$, we have:*

- *If Σ is ground, then $\lceil\Theta\rceil;\lceil\Xi\rceil;\lceil\Gamma\rceil \vdash \lceil c\rceil : \lceil A\rceil\,!\,\lceil\Sigma\rceil$ and $\lceil\Theta\rceil;\lceil\Xi\rceil;\lceil\Gamma\rceil \vdash \lceil v\rceil : \lceil A\rceil$.*
- *$\lfloor\Theta\rfloor;\lfloor\Xi\rfloor;\lfloor\Gamma\rfloor \vdash \lceil c\rceil : \lfloor A\rfloor$ and $\lfloor\Theta\rfloor;\lfloor\Xi\rfloor;\lfloor\Gamma\rfloor \vdash \lceil v\rceil : \lfloor A\rfloor$.*

Proof
For the first part only, first show that if A is ground, then $\lceil A\rceil = A$, and so if Σ is a well-kinded ground signature, then $\lceil\Sigma\rceil$ is well-defined and well-kinded.

Then the proofs of both parts follow the same lines. By mutual induction on the kinding judgements, show that well-kinded types, schemes, and contexts translate into well-kinded types, schemes, and contexts, respectively. Then show that both translations respect type-level substitution:

$$\lceil B[A_i/\alpha_i]_{1\le i\le n}\rceil = \lceil B\rceil[\lceil A_i\rceil/\alpha_i]_{1\le i\le n}$$

and similarly for the coarse translation.

Finally, by mutual induction on typing judgements for values and computations, and on scheming judgements, show the hypothesis. We mention only the interesting cases.

For dereferencing a cell $(p : A) \in \Sigma$, by the translation's definition,

$$(\mathtt{get\_}p : \mathsf{unit} \to \lceil A\rceil) \in \lceil\Sigma\rceil$$

Use this fact to derive that $\lceil !p\rceil$ has the type $\lceil A\rceil$. Use a similar argument for assignment.

Next, show that for all $(p : A) \in \Sigma$:

$$\lceil\Theta\rceil;\lceil\Xi\rceil;\lceil\Gamma\rceil \vdash H^{p}_{ST} : (B\,!\lceil\Sigma\rceil) \Rightarrow ((\lceil A\rceil \to (B\,!\lceil\Sigma\rceil))\,!\lceil\Sigma\rceil)$$

and use this fact, together with the induction hypotheses, to give a valid derivation for $\lceil\mathbf{dlet}\, p \leftarrow v\ \mathbf{in}\ c\rceil$. ■

In summary, the handler H_{ST} expresses dynamically scoped state, in both terms and types.

5 Related work

In addition to all above discussions of immediately relevant work, we now provide the interested reader with a short survey of existing literature in related areas.

Polymorphism and type inference. The System F of Girard (1972) and the polymorphic λ-calculus of Reynolds (1974) pioneer the meta-theory of polymorphic computation to which we contribute. The impredicativity of their proposed systems has two consequences relevant to our setting. First, because polymorphic types can appear anywhere a type may appear, type inference becomes undecidable (Wells, 1999). Second, because universally quantified type variables range over the types which they are used to define, these calculi have no set-theoretic models (Reynolds, 1984). Hindley-Milner polymorphism (Milner, 1978) avoids these two shortcomings (Harper & Mitchell, 1993; Ohori, 1989), providing a convenient theoretical and practical setting for investigating the integration of effects and polymorphism. Row polymorphism (Wand, 1987; Ohori, 1995, 1992; Garrigue, 2001, 2010, 2015) allows a polymorphic treatment for extensible records, or labelled products. The salient features of such records and their operations (Cardelli & Mitchell, 1991; Harper & Pierce, 1991) is their contribution to program modularity and their compatibility with Hindley-Milner polymorphism (Rémy, 1990, 1991), which may be relevant for programming with algebraic effects and handlers (Hillerström & Lindley, 2016).

The effect polymorphism problem and the value restriction. Gordon *et al.* (1979) first describe the problem under consideration in the context of integrating references with Hindley-Milner polymorphism. Harper & Lillibridge (1993b) noticed the same issue arises with the control operators **call/cc** and **throw** and announced it on the `types` mailing list in July 1991. There have been many proposed solutions, notably Leroy (1992, 1993); Leroy & Weis (1991); Tofte (1990) and Appel & MacQueen's (1991) Standard ML of New Jersey. Wright (1995) argues for the sufficiency of the value restriction in practical implementations, and Garrigue (2004) argues for its relaxation. Wright's solution has become standard undergraduate textbook material (Pierce, 2002; Rémy, 2015; Pitts, 2011–2016). Nonetheless, Kiselyov (2015) advocates relaxing the value restriction even further to facilitate implementation techniques for staged computation. Our contribution demonstrates such relaxation is possible in the algebraic case. Our work originates from a semantic analysis of the interaction of effects and polymorphism. In this vein, Zeilberger (2009); Munch-Maccagnoni (2009); and Lepigre (2016) propose semantic perspectives on the value restriction using focussed calculi and realisability semantics, and Jaber & Tzevelekos (2016) propose game semantics as an investigatory tool.

Effect systems. The effect system plays a central role in the typing of algebraic effects in general, and in particular in our system. Lucassen & Gifford (1988) introduce a polymorphic effect system to improve execution times using implicit parallelism in the context of the *FX* programming language. While the original system only kept track of region-based memory accesses and allocation, it was clear the methodology can be adapted to multiple kinds of effects and analyses, including: control-flow analysis, binding-time analysis, and process communication, see Nielson & Nielson (1999) for a survey. The nature of the effect annotations can be descriptive, over-approximating which effects *may* be caused, or more intensional and prescriptive, tracking exactly the order in which the effects must occur. Wadler & Thiemann (2003); Tolmach (1998); and Benton *et al.* (1998) independently make the connection between type-and-effect systems and multi-monadic type systems, marking the beginning of a general theory. Marino & Millstein (2009) propose a general framework for effect systems, and Kammar & Plotkin (2012); Kammar (2014); and Katsumata (2014) propose general theories of effect systems that include a denotational semantic account, which underlies the system we propose. Leroy & Weis's (1991) use of an effect system to address the effect polymorphism problem is very close to ours. Effect systems have also been used by Rompf *et al.* (2009) to enable a more efficient compilation of delimited control from Scala to Java bytecodes, and by Lippmeier's (2009) compiler for the *Disciple* language during the optimisation process and to mix lazy and strict evaluation order. In the context of algebraic effects and handlers, Bauer & Pretnar (2014) propose an effect system with sub-effecting, and Hillerström & Lindley (2016) propose an effect system with effect polymorphism based on on their implementation of row types in the Links web programming language (Cooper *et al.*, 2006; Lindley & Cheney, 2012).

The algebraic theory of computational effects. Algebraic effects and handlers arise from the monadic and algebraic account of computational effects, which we survey briefly here. Moggi (1991) conceptualises computational effects using monads. Plotkin & Power (2003, 2002) refine this account by incorporating the syntactic constructs causing the effects into the meta-theory in terms of equations between universal algebraic terms. The algebraic account allows a more refined analysis of existing effects (Staton, 2010, 2009, 2013b; Melliès, 2014, 2010) and new effects (Staton, 2013a, 2015; Fiore & Staton, 2014). The inclusion of effect operations into the meta-theory faciliates a broader development accounting for: effect combination (Hyland *et al.*, 2006), extended program logics (Plotkin & Pretnar, 2008; Pretnar, 2010), effect-dependent program transformations (Kammar & Plotkin, 2012; Kammar, 2014), monadic lifting of logical relations (Katsumata, 2013), dependent types (Ahman *et al.*, 2016) and refinement types (Ahman, 2013), and normalization-by-evaluation (Ahman & Staton, 2013).

Programming with algebraic effects and handlers. Several researchers simultaneously advocated the use of the handler programming abstraction. In analogy with Wadler's (1992) use of monads to structure imperative functional programming, Bauer & Pretnar (2015) advocate a similar use of algebraic effects and handlers in the programming language *Eff*, an *ML*-style language with type inference (Pretnar, 2014). Kiselyov *et al.* (2013) and Kiselyov & Ishii (2015) implement an effect handlers library for *Haskell*, based on Cartwright & Felleisen's (1994) work on extensible denotational semantics and interpreters. Kammar *et al.* (2013), in addition to *Haskell* implementations based on continuations and free monads (Swierstra, 2008; Hancock & Setzer, 2000), implement handler libraries in *OCaml*, *Standard-ML*, and *Racket*, and give a sound small-step operational semantics for a core calculus of handlers with local effect signatures, which underlies the semantics we give here. Handlers integrate smoothly with other program features: Brady's (2013, 2014) effect handler library in the *Idris* language uses dependent-types to reason about effectful code, and Saleh & Schrijvers (2016) add effect handlers to the *Prolog* language as a high-level alternative to delimited control. Wu *et al.* (2014) propose a generalisation to effect handlers that allows scoped effects, which seems to increase their expressiveness. While we deal with the semantic foundations of handlers, there is ongoing investigation into their implementation. Wu & Schrijvers (2015) analyse and explain the runtime efficiency of implementation techniques for effect handlers, and in particular, free monads. The *Koka* language of Leijen (2014, 2017) offers run-time and compilation support for algebraic effects and handlers, using a selective continuation-passing-style transformation. McBride investigates untyped effect handlers through the *Shonky* language[3] and variations on polymorphic effect systems through the *Frank* language (Lindley *et al.*, 2017). Dolan, White, Sivaramakrishnan, Yallop, and Madhavapeddy recently propose[4] a language and runtime extension to the *OCaml* language to support algebraic effects.

Delimited control. We also view handlers as a delimited control operator. Felleisen (1988) first introduces control delimiters as a mechanism for improving the meta-theory of control operators. Early on (Felleisen *et al.*, 1988), they were used to implement algorithms with sophisticated control structure, such as tree-fringe comparison, and other control mechanisms, such as coroutines. Danvy & Filinski (1990) study control operators systematically using continuation-passing-style conversion, and introduce a hierarchy of control operators, including the operators **shift** and $\mathbf{shift}_0$, the latter being the traditional control operator deep handlers are closest to. Danvy & Filinski (1989) give the first well-known type system for delimited control. The different variations in control structure and the desire to relax the type system to accept more sophisticated programs involving answer-type modification, lead to a plethora of type systems for delimited control, and Kiselyov & Shan (2007) give a substantial list of references. The most relevant to this work are the polymorphic type systems of Gunter *et al.* (1995); Kiselyov *et al.* (2006); and Asai & Kameyama (2007). The expressive relationship between different delimited control constructs is subtle, and Shan (2007) establishes some untyped relationships using Felleisen's (1991) notion of macro expressibility. In collaboration with Forster *et al.* (2016), we similarly investigate the expressiveness of handlers with delimited control and monadic reflection. Danvy (2006) and Kiselyov *et al.* (2006) show how delimited control operators can simulate global state and Moreau's (1998) notion of dynamically scoped state, respectively, on which we base our characterisation of the H_{ST} handler.

Expressing recursion with delimited control. The ability to encode Landin's (1964) knot, while possible with handlers and coarse type annotation, is already recurrent in the study of control operators. Lillibridge (1999) shows unchecked exceptions are Turing-complete by defining the Y-combinator. Filinski's (1994) type system for **shift** and **reset** fixes the answer type, and it is well-known folklore that if the answer type is a function

[3] `https://github.com/pigworker/shonky`
[4] Talk given at the OCaml Workshop 2015.

type, the calculus is non-terminating. Consequently, Filinski's continuation-passing-style translation into the simply typed lambda calculus requires recursive types. Kameyama & Yonezawa's (2008) simple type system for Felleisen's (1988) delimited-control operators **control** and **prompt** can similarly tie the knot.

6 Conclusion and further work

Unexpectedly, Hindley-Milner polymorphism integrates smoothly and robustly with existing type-and-effect systems for algebraic effects and handlers. However, combining reference cell allocation with polymorphism remains an open problem, as does incorporating dynamic generation of instances as used in *Eff*. Consequently, *Eff* still uses the value restriction. Our contribution is to identify a larger class of languages in which effects and polymorphism coexist naturally.

For type-system cognoscenti, these results may not come as a complete surprise. First, using effect systems to ensure soundness has been proposed (Leroy & Weis, 1991) before Wright's value restriction. Second, even if we consider the non-effect-annotated safety result, we do not believe the type system can encode the problematic effects: local reference cells and continuations. Nonetheless, previous solutions require a *specialised*, and sometimes subtle, type system. In the algebraic setting, adding polymorphism to existing systems is strikingly natural.

There is an intuitive explanation for polymorphism-by-name and the value restriction using an analogy between the polymorphic type abstraction operation, i.e. Λ in System F (Girard, 1972; Reynolds, 1974), and the function abstraction operation, i.e., λ in the λ-calculus, which is already present in, e.g. Leroy (1992, 1993); Cardelli (1991); Gifford & Lucassen (1986); Wright (1995); and Harper & Lillibridge (1993a). By elaborating the types of bound variables and inserting an explicit type abstraction operation to the example in the introduction, we get:

$$\begin{array}{ll}\textbf{let}\ id = \Lambda\alpha.(\textbf{fun}\ f : (\alpha \to \alpha) \to \alpha \to \alpha \mapsto f)\ (\textbf{fun}\ x : \alpha \mapsto x)\ \textbf{in} & (*\forall\alpha.\alpha \to \alpha*) \\ id\,(id) & \end{array}$$

The λ-abstraction operation suspends computation in the λ-calculus, and so, by analogy, we may choose to suspend computation under Λ-abstraction as well. Doing so results in Leroy's polymorphism by name. If we want to keep a call-by-value evaluation order on polymorphic let and treat Λ as computation suspending, the value restriction is the natural solution.

However, Garrigue's relaxation of the value restriction and its implementation in OCaml show that programmers are interested in separating suspension of computation from type abstraction, departing from the behaviour for function abstraction. Moreover, Levy's (2004) call-by-push-value shows that functional calculi for effects, and their resulting equational theories, benefit from a lingual separation between suspension of computation, i.e., thunking, and functional abstraction. In this situation, we are interested to know how to integrate call-by-value polymorphism with effects.

An anonymous reviewer suggests the following intuition to our result in terms of *sharing* (Leroy, 1992). Reference cells allow creating some shared state without exposing its type in the shared context. As a result, without the value restriction, it becomes possible to generalise this type wrongly. In contrast, our type system is fully explicit. Any state manipulation, and more generally, any effect manipulation, is through the handlers via the operation signatures, whose types are either declared globally (in the coarser system) or explicitly. The cause for this explicitness is clear: unlike state manipulation operations for reference cells whose meaning is fixed, algebraic effect operations are decoupled from their concrete semantics. Thus we need to track types explicitly so that the enclosing handler uses the correct types. As a consequence, the hidden sharing is lost, and there is no need to restrict polymorphism. But by losing the ability to hide shared state, we can no longer easily express reference cells.

We have also shown formally that the 'local' state handler H_{ST} simulates dynamically scoped state. In particular, our type-preserving translation establishes that, like algebraic effects, dynamically scoped state does not need a value restriction. We contrast this result with one of Wright's motivations for the value restriction — providing a safe type system for an imperative language with no additional annotation on function types. Therefore, as far as dynamically scoped state is concerned, the unannotated polymorphic calculus for effect handlers demonstrates there is no need for the value restriction to achieve Wright's goal.

These results arose as part of a broader (denotational) semantic investigation of effects and polymorphism, which does not yet account for reference cells. We hope that an algebraic understanding of locality (Staton, 2013b; Fiore & Staton, 2014) and scope and polymorphic arities (Wu *et al.*, 2014) will explain the interaction between reference cells and polymorphism. As mentioned in the end of Sec. 3, our proof easily adapts to the presence of effect instances, though only for global signatures. The question of how to combine effect instances with local annotations is still open, and will most likely also be answered by understanding locality. The robustness of type safety leads us to believe standard extensions, such as type inference, principal types, and impredicative and row polymorphism will not pose problems. The latter is particularly interesting, as it can serve as an effect system with effect variables (Lindley & Cheney, 2012; Hillerström & Lindley, 2016; Leijen, 2014; Pretnar, 2014).

We want to investigate the expressive difference between effect handlers and delimited control, and polymorphism forms another comparison axis. We defer a thorough comparison, as there are several notions of delimited control (**shift**, $\mathbf{shift}_0$, with or without answer-type modification) and several proposals for polymorphic type systems (Asai & Kameyama, 2007; Gunter *et al.*, 1995; Kiselyov *et al.*, 2006), and as delimited control is subtle. That said, there are two immediate points of comparison between delimited control and effect handlers.

First, Kiselyov *et al.*'s translation of dynamic scope into delimited control requires some complication in order to preserve types. This complication is caused by their effect system for delimited control tracking the return type of the computation enclosed by the nearest rebinding. When an access to a dynamically scoped cell escapes the current binding in scope, the type expected in the nearest rebinding may change, resulting in a type error of their translated program. The example on page 16 demonstrates such a shift from a function type to an integer type. In contrast, our effect system only tracks the local type for each effect operation, and the translation from dynamically scoped state to effect handlers extends smoothly to types.

Second, these systems include a form of a purity restriction or value restriction. As a consequence, they cannot type purely functional computations like the final example in Subsection 4.1. In contrast, the type system proposed here allows unrestricted Hindley-Milner polymorphism in both purely functional and effectful code.

Acknowledgements. We thank Alex Simpson for conjecturing the value restriction may not be necessary for algebraic effects. We also thank the Journal of Functional Programming editors Matthias Felleisen, François Pottier, and Jeremy Gibbons, and the anonymous referees, whose hard work improved this article considerably. We also thank Andrej Bauer, Xavier Leroy, Sam Lindley, Sean Moss, Dominic Mulligan, Jean Pichon-Pharabod, Andy Pitts, Didier Rémy, Philip Wadler, and Jeremy Yallop for many useful suggestions and discussions.

Bibliography

Ahman, Danel. (2013). Refinement types and algebraic effects. *2nd Workshop on Higher-Order Programming with Effects, HOPE 2013.*

Ahman, Danel, & Staton, Sam. (2013). Normalization by evaluation and algebraic effects. *Electr. notes theor. comput. sci.*, **298**, 51–69.

Ahman, Danel, Ghani, Neil, & Plotkin, Gordon D. (2016). Dependent types and fibred computational effects. *Pages 36–54 of:* Jacobs, Bart, & Löding, Christof (eds), *Foundations of Software Science and Computation Structures - 19th International Conference, FOSSACS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Lecture Notes in Computer Science, vol. 9634. Springer.

Appel, Andrew W., & MacQueen, David B. (1991). Standard ML of New Jersey. *Pages 1–13 of: PLILP.*

Asai, Kenichi, & Kameyama, Yukiyoshi. (2007). Polymorphic delimited continuations. *Pages 239–254 of: APLAS.* Lecture Notes in Computer Science, vol. 4807. Springer.

Bauer, Andrej, & Pretnar, Matija. (2014). An effect system for algebraic effects and handlers. *Logical methods in computer science*, **10**(4).

Bauer, Andrej, & Pretnar, Matija. (2015). Programming with algebraic effects and handlers. *J. log. algebr. meth. program.*, **84**(1), 108–123.

Benton, Nick, Kennedy, Andrew, & Russell, George. (1998). Compiling Standard ML to Java Bytecodes. *Pages 129–140 of:* Felleisen, Matthias, Hudak, Paul, & Queinnec, Christian (eds), *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98), Baltimore, Maryland, USA, September 27-29, 1998.* ACM.

Brady, Edwin. (2013). Programming and reasoning with algebraic effects and dependent types. *Pages 133–144 of:* Morrisett, Greg, & Uustalu, Tarmo (eds), *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013.* ACM.

Brady, Edwin. (2014). Resource-dependent algebraic effects. *Pages 18–33 of:* Hage, Jurriaan, & McCarthy, Jay (eds), *Trends in Functional Programming - 15th International Symposium, TFP 2014, Soesterberg, The Netherlands, May 26-28, 2014. Revised Selected Papers.* Lecture Notes in Computer Science, vol. 8843. Springer.

Cardelli, Luca. (1991). Typeful programming. *Pages 431–507 of:* Neuhold, E. J., & Paul, M. (eds), *Formal Description of Programming Concepts*. Berlin: Springer-Verlag.

Cardelli, Luca, & Mitchell, John C. (1991). Operations on records. *Mathematical structures in computer science*, **1**(1), 3–48.

Cartwright, Robert, & Felleisen, Matthias. (1994). Extensible denotational language specifications. *Pages 244–272 of:* Hagiya, Masami, & Mitchell, John C. (eds), *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*. Lecture Notes in Computer Science, vol. 789. Springer.

Cooper, Ezra, Lindley, Sam, Wadler, Philip, & Yallop, Jeremy. (2006). Links: Web programming without tiers. *Pages 266–296 of:* de Boer, Frank S., Bonsangue, Marcello M., Graf, Susanne, & de Roever, Willem P. (eds), *Formal Methods for Components and Objects, 5th International Symposium, FMCO 2006, Amsterdam, The Netherlands, November 7-10, 2006, Revised Lectures*. Lecture Notes in Computer Science, vol. 4709. Springer.

Danvy, Olivier. (2006). *An analytical approach to programs as data objects*. DSc dissertation, Department of Computer Science, University of Aarhus.

Danvy, Olivier, & Filinski, Andrzej. (1989). *A functional abstraction of typed contexts*. Tech. rept. 89/12. DIKU.

Danvy, Olivier, & Filinski, Andrzej. (1990). Abstracting control. *Pages 151–160 of: LISP and Functional Programming*.

Felleisen, Matthias. (1988). The theory and practice of first-class prompts. *Pages 180–190 of:* Ferrante, Jeanne, & Mager, P. (eds), *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*. ACM Press.

Felleisen, Matthias. (1991). On the expressive power of programming languages. *Sci. comput. program.*, **17**(1-3), 35–75.

Felleisen, Matthias, Wand, Mitchell, Friedman, Daniel P., & Duba, Bruce F. (1988). Abstract continuations: A mathematical semantics for handling full jumps. *Pages 52–62 of: LISP and Functional Programming*.

Filinski, Andrzej. (1994). Representing monads. *Pages 446–457 of:* Boehm, Hans-Juergen, Lang, Bernard, & Yellin, Daniel M. (eds), *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*. ACM Press.

Fiore, Marcelo P., & Staton, Sam. (2014). Substitution, jumps, and algebraic effects. *Pages 41:1–41:10 of: CSL-LICS*. ACM.

Forster, Yannick, Kammar, Ohad, Lindley, Sam, & Pretnar, Matija. (2016). On the expressive power of user-defined effects: Effect handlers, monadic reflection, delimited control. *Corr*, **abs/1610.09161**.

Garrigue, Jacques. (2001). Simple type inference for structural polymorphism. *Pages 329–343 of: The Second Asian Workshop on Programming Languages and Systems, APLAS'01, Korea Advanced Institute of Science and Technology, Daejeon, Korea, December 17-18, 2001, Proceedings*.

Garrigue, Jacques. (2004). Relaxing the value restriction. *Pages 196–213 of: FLOPS*. Lecture Notes in Computer Science, vol. 2998. Springer.

Garrigue, Jacques. (2010). A certified implementation of ML with structural polymorphism. *Pages 360–375 of:* Ueda, Kazunori (ed), *Programming Languages and Systems - 8th Asian Symposium, APLAS 2010, Shanghai, China, November 28 - December 1, 2010. Proceedings.* Lecture Notes in Computer Science, vol. 6461. Springer.

Garrigue, Jacques. (2015). A certified implementation of ML with structural polymorphism and recursive types. *Mathematical structures in computer science*, **25**(4), 867–891.

Gifford, David K., & Lucassen, John M. (1986). Integrating functional and imperative programming. *Pages 28–38 of: LISP and Functional Programming.*

Girard, Jean-Yves. 1972 (June). *Interprétation fonctionnelle et Élimination des coupures de l'arithmétique d'ordre supérieur.* Thèse de doctorat d'état, Université Paris VII.

Gordon, Andrew D. (ed). (2017). *Fourty-fourth annual ACM symposium on principles of programming languages, paris, france,* to appear. ACM Press.

Gordon, Michael J. C., Milner, Robin, & Wadsworth, Christopher P. (1979). *Edinburgh LCF.* Lecture Notes in Computer Science, vol. 78. Springer.

Gunter, Carl A., Rémy, Didier, & Riecke, Jon G. (1995). A generalization of exceptions and control in ML-like languages. *Pages 12–23 of: FPCA.* ACM.

Hancock, Peter, & Setzer, Anton. (2000). Interactive programs in dependent type theory. *Pages 317–331 of:* Clote, Peter, & Schwichtenberg, Helmut (eds), *Computer Science Logic, 14th Annual Conference of the EACSL, Fischbachau, Germany, August 21-26, 2000, Proceedings.* Lecture Notes in Computer Science, vol. 1862. Springer.

Harper, Robert, & Lillibridge, Mark. (1993a). Explicit polymorphism and CPS conversion. *Pages 206–219 of:* Deusen, Mary S. Van, & Lang, Bernard (eds), *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993.* ACM Press.

Harper, Robert, & Lillibridge, Mark. (1993b). Polymorphic type assignment and CPS conversion. *Lisp and symbolic computation*, **6**(3-4), 361–380.

Harper, Robert, & Mitchell, John C. (1993). On the type structure of standard ML. *ACM trans. program. lang. syst.*, **15**(2), 211–252.

Harper, Robert, & Pierce, Benjamin C. (1991). A record calculus based on symmetric concatenation. *Pages 131–142 of:* Wise, David S. (ed), *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991.* ACM Press.

Hillerström, Daniel, & Lindley, Sam. (2016). Liberating effects with rows and handlers. *Pages 15–27 of:* Chapman, James, & Swierstra, Wouter (eds), *Proceedings of the 1st International Workshop on Type-Driven Development, September 2016.* ACM.

Hyland, Martin, Plotkin, Gordon D., & Power, John. (2006). Combining effects: Sum and tensor. *Theor. comput. sci.*, **357**(1-3), 70–99.

Jaber, Guilhem, & Tzevelekos, Nikos. (2016). Trace semantics for polymorphic references. *Corr*, **abs/1602.08406**.

Kameyama, Yukiyoshi, & Yonezawa, Takuo. (2008). Typed dynamic control operators for delimited continuations. *Pages 239–254 of:* Garrigue, Jacques, & Hermenegildo, Manuel V. (eds), *Functional and Logic Programming, 9th International Symposium, FLOPS 2008, Ise, Japan, April 14-16, 2008. Proceedings.* Lecture Notes in Computer Science, vol. 4989. Springer.

Kammar, Ohad. (2014). *An algebraic theory of type-and-effect systems*. Ph.D. thesis, University of Edinburgh, UK.

Kammar, Ohad, & Plotkin, Gordon D. (2012). Algebraic foundations for effect-dependent optimisations. *Pages 349–360 of:* Field, John, & Hicks, Michael (eds), *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. ACM.

Kammar, Ohad, Lindley, Sam, & Oury, Nicolas. (2013). Handlers in action. *Pages 145–158 of: ICFP*. ACM.

Katsumata, Shin-ya. (2013). Relating computational effects by $\top\top$-lifting. *Inf. comput.*, **222**, 228–246.

Katsumata, Shin-ya. (2014). Parametric effect monads and semantics of effect systems. *Pages 633–646 of:* Jagannathan, Suresh, & Sewell, Peter (eds), *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM.

Kiselyov, Oleg. 2015 (September). *Generating code with polymorphic let*. Tech. rept. Tohoku University, Japan. extended abstract submitted to the ACM SIGPLAN Workshop on ML.

Kiselyov, Oleg, & Ishii, Hiromi. (2015). Freer monads, more extensible effects. *Pages 94–105 of:* Lippmeier, Ben (ed), *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*. ACM.

Kiselyov, Oleg, & Shan, Chung-chieh. (2007). A substructural type system for delimited continuations. *Pages 223–239 of:* Rocca, Simona Ronchi Della (ed), *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*. Lecture Notes in Computer Science, vol. 4583. Springer.

Kiselyov, Oleg, Shan, Chung-chieh, & Sabry, Amr. (2006). Delimited dynamic binding. *Pages 26–37 of: ICFP*. ACM.

Kiselyov, Oleg, Sabry, Amr, & Swords, Cameron. (2013). Extensible effects: an alternative to monad transformers. *Pages 59–70 of: Haskell*. ACM.

Landin, P. J. (1964). The mechanical evaluation of expressions. *The computer journal*, **6**(4), 308–320.

Leijen, Daan. (2014). Koka: Programming with row polymorphic effect types. *Pages 100–126 of: MSFP*. EPTCS, vol. 153.

Leijen, Daan. (2017). Type directed compilation of row-typed algebraic effects. *In:* Gordon (2017).

Lepigre, Rodolphe. (2016). A classical realizability model for a semantical value restriction. *Pages 476–502 of:* Thiemann, Peter (ed), *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Lecture Notes in Computer Science, vol. 9632. Springer.

Leroy, Xavier. (1992). *Typage polymorphe d'un langage algorithmique*. PhD thesis (in French), Université Paris 7.

Leroy, Xavier. (1993). Polymorphism by name for references and continuations. *Pages 220–231 of: POPL*. ACM Press.

Leroy, Xavier, & Weis, Pierre. (1991). Polymorphic type inference and assignment. *Pages 291–302 of: POPL*. ACM Press.

Levy, Paul B. (2004). *Call-by-push-value: A functional/imperative synthesis*. Semantics Structures in Computation, vol. 2. Springer.

Levy, Paul Blain, Power, John, & Thielecke, Hayo. (2003). Modelling environments in call-by-value programming languages. *Inf. comput.*, **185**(2), 182–210.

Lillibridge, Mark. (1999). Unchecked exceptions can be strictly more powerful than call/cc. *Higher-order and symbolic computation*, **12**(1), 75–104.

Lindley, Sam, & Cheney, James. (2012). Row-based effect types for database integration. *Pages 91–102 of: TLDI*. ACM.

Lindley, Sam, McBride, Conor, & McLaughlin, Craig. (2017). Do be do be do. *In:* Gordon (2017).

Lippmeier, Ben. (2009). Witnessing purity, constancy and mutability. *Pages 95–110 of:* Hu, Zhenjiang (ed), *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. Lecture Notes in Computer Science, vol. 5904. Springer.

Lucassen, John M., & Gifford, David K. (1988). Polymorphic effect systems. *Pages 47–57 of: POPL*. ACM Press.

Marino, Daniel, & Millstein, Todd D. (2009). A generic type-and-effect system. *Pages 39–50 of:* Kennedy, Andrew, & Ahmed, Amal (eds), *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*. ACM.

Melliès, Paul-André. (2010). Segal condition meets computational effects. *Pages 150–159 of: Proceedings of the 25th Annual IEEE Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom*. IEEE Computer Society.

Melliès, Paul-André. (2014). Local states in string diagrams. *Pages 334–348 of:* Dowek, Gilles (ed), *Rewriting and Typed Lambda Calculi - Joint International Conference, RTA-TLCA 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*. Lecture Notes in Computer Science, vol. 8560. Springer.

Milner, Robin. (1978). A theory of type polymorphism in programming. *J. comput. syst. sci.*, **17**(3), 348–375.

Moggi, Eugenio. (1991). Notions of computation and monads. *Inf. comput.*, **93**(1), 55–92.

Moreau, Luc. (1998). A syntactic theory of dynamic binding. *Higher-order and symbolic computation*, **11**(3), 233–279.

Munch-Maccagnoni, Guillaume. (2009). Focalisation and classical realisability. *Pages 409–423 of:* Grädel, Erich, & Kahle, Reinhard (eds), *Computer Science Logic '09*. Lecture Notes in Computer Science, vol. 5771. Springer, Heidelberg.

Nielson, Flemming, & Nielson, Hanne Riis. (1999). Type and effect systems. *Pages 114–136 of:* Olderog, Ernst-Rüdiger, & Steffen, Bernhard (eds), *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*. Lecture Notes in Computer Science, vol. 1710. Springer.

Ohori, Atsushi. (1989). A simple semantics for ML polymorphism. *Pages 281–292 of:* Stoy, Joseph E. (ed), *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*. ACM.

Ohori, Atsushi. (1992). A compilation method for ML-style polymorphic record calculi. *In:* Sethi (1992).

Ohori, Atsushi. (1995). A polymorphic record calculus and its compilation. *ACM trans. program. lang. syst.*, **17**(6), 844–895.

Pfenning, Frank, & Schürmann, Carsten. (1999). System description: Twelf - A meta-logical framework for deductive systems. *Pages 202–206 of: CADE*. Lecture Notes in Computer Science, vol. 1632. Springer.

Pierce, Benjamin C. (2002). *Types and programming languages*. Cambridge, MA, USA: MIT Press.

Pitts, Andrew M. (2011–2016). *Types*. Lecture notes. University of Cambridge Computer Laboratory.

Plotkin, Gordon D. (1977). LCF considered as a programming language. *Theor. comput. sci.*, **5**(3), 223–255.

Plotkin, Gordon D., & Power, John. (2002). Notions of computation determine monads. *Pages 342–356 of:* Nielsen, Mogens, & Engberg, Uffe (eds), *Foundations of Software Science and Computation Structures, 5th International Conference, FOSSACS 2002. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002 Grenoble, France, April 8-12, 2002, Proceedings*. Lecture Notes in Computer Science, vol. 2303. Springer.

Plotkin, Gordon D., & Power, John. (2003). Algebraic operations and generic effects. *Applied categorical structures*, **11**(1), 69–94.

Plotkin, Gordon D., & Pretnar, Matija. (2008). A logic for algebraic effects. *Pages 118–129 of: Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA*. IEEE Computer Society.

Plotkin, Gordon D., & Pretnar, Matija. (2013). Handling algebraic effects. *Logical methods in computer science*, **9**(4).

Pretnar, Matija. (2010). *Logic and handling of algebraic effects*. Ph.D. thesis, University of Edinburgh, UK.

Pretnar, Matija. (2014). Inferring algebraic effects. *Logical methods in computer science*, **10**(3).

Pretnar, Matija. (2015). An introduction to algebraic effects and handlers. invited tutorial paper. *Electr. notes theor. comput. sci.*, **319**, 19–35.

Rémy, Didier. (1990). *Algèbres touffues. application au typage polymorphe des objets enregistrements dans les langages fonctionnels*. Thèse de doctorat, Université de Paris 7.

Rémy, Didier. 1991 (May). *Type inference for records in a natural extension of ML*. Research Report 1431. Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France.

Rémy, Didier. (2015). *Type systems*. Lecture notes. Parisian Master of Research in Computer Science.

Reynolds, John C. (1974). Towards a theory of type structure. *Pages 408–423 of: Programming Symposium, Proceedings Colloque Sur La Programmation*. London, UK, UK: Springer-Verlag.

Reynolds, John C. (1984). Polymorphism is not set-theoretic. *Pages 145–156 of:* Kahn, Gilles, MacQueen, David B., & Plotkin, Gordon D. (eds), *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*. Lecture Notes in Computer Science, vol. 173. Springer.

Rompf, Tiark, Maier, Ingo, & Odersky, Martin. (2009). Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *Pages 317–328 of:* Hutton, Graham, & Tolmach, Andrew P. (eds), *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*. ACM.

Saleh, Amr Hany, & Schrijvers, Tom. (2016). Efficient algebraic effect handlers for prolog. *Corr*, **abs/1608.00816**.

Scott, Dana S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theor. comput. sci.*, **121**(1&2), 411–440.

Sethi, Ravi (ed). (1992). *Conference record of the nineteenth annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, albuquerque, new mexico, usa, january 19-22, 1992*. ACM Press.

Shan, Chung-chieh. (2007). A static simulation of dynamic delimited control. *Higher-order and symbolic computation*, **20**(4), 371–401.

Staton, Sam. (2009). Two cotensors in one: Presentations of algebraic theories for local state and fresh names. *Electr. notes theor. comput. sci.*, **249**, 471–490.

Staton, Sam. (2010). Completeness for algebraic theories of local state. *Pages 48–63 of:* Ong, C.-H. Luke (ed), *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*. Lecture Notes in Computer Science, vol. 6014. Springer.

Staton, Sam. (2013a). An algebraic presentation of predicate logic - (extended abstract). *Pages 401–417 of:* Pfenning, Frank (ed), *Foundations of Software Science and Computation Structures - 16th International Conference, FOSSACS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Lecture Notes in Computer Science, vol. 7794. Springer.

Staton, Sam. (2013b). Instances of computational effects: An algebraic perspective. *Page 519 of: LICS*. IEEE Computer Society.

Staton, Sam. (2015). Algebraic effects, linearity, and quantum programming languages. *Pages 395–406 of:* Rajamani, Sriram K., & Walker, David (eds), *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. ACM.

Swierstra, Wouter. (2008). Data types à la carte. *J. funct. program.*, **18**(4), 423–436.

Tofte, Mads. (1990). Type inference for polymorphic references. *Inf. comput.*, **89**(1), 1–34.

Tolmach, Andrew P. (1998). Optimizing ML using a hierarchy of monadic types. *Pages 97–115 of:* Leroy, Xavier, & Ohori, Atsushi (eds), *Types in Compilation, Second International Workshop, TIC '98, Kyoto, Japan, March 25-27, 1998, Proceedings*. Lecture Notes in Computer Science, vol. 1473. Springer.

Wadler, Philip. (1992). The essence of functional programming. *In:* Sethi (1992).

Wadler, Philip, & Thiemann, Peter. (2003). The marriage of effects and monads. *ACM trans. comput. log.*, **4**(1), 1–32.

Wand, Mitchell. (1987). Complete type inference for simple objects. *Pages 37–44 of: Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*. IEEE Computer Society.

Wells, J. B. (1999). Typability and type checking in System F are equivalent and undecidable. *Ann. pure appl. logic*, **98**(1-3), 111–156.

Wright, Andrew K. (1995). Simple imperative polymorphism. *Lisp and symbolic computation*, **8**(4), 343–355.

Wu, Nicolas, & Schrijvers, Tom. (2015). Fusion for free - efficient algebraic effect handlers. *Pages 302–322 of:* Hinze, Ralf, & Voigtländer, Janis (eds), *Mathematics of Program Construction - 12th International Conference, MPC 2015, Königswinter, Germany, June 29 - July 1, 2015. Proceedings*. Lecture Notes in Computer Science, vol. 9129. Springer.

Wu, Nicolas, Schrijvers, Tom, & Hinze, Ralf. (2014). Effect handlers in scope. *Pages 1–12 of: Haskell*. ACM.

Zeilberger, Noam. (2009). Refinement types and computational duality. *Pages 15–26 of:* Altenkirch, Thorsten, & Millstein, Todd D. (eds), *Proceedings of the 3rd ACM Workshop Programming Languages meets Program Verification, PLPV 2009, Savannah, GA, USA, January 20, 2009*. ACM.