Dylan McDermott* and Alan Mycroft

Call-by-need effects via coeffects

Abstract: Effect systems refine types with information about the behaviour of programs. They have been used for many purposes, such as optimizing programs, determining resource usage, and finding bugs. So far, however, work on effect systems has largely concentrated on call-by-value languages.
We consider the problem of designing an effect system for a lazy language. This is more challenging because it depends on the ability to locate the first use of each variable. Coeffect systems, which track contextual requirements of programs, provide a method of doing this. We describe how to track variable usage in a coeffect system that can be instantiated for different reduction strategies, including call-by-need. We then add effects to the result, allowing work that has been done on effect systems for call-by-value languages to be applied to lazy languages.

Keywords: Effect systems, coeffect systems, call-by-need

1 Introduction

Effect systems refine type systems so that they statically estimate the side-effects of expressions. Along with the type, the typing judgement also contains an effect, which is an abstract value that gives an upper bound on the side-effect of the expression. Many effect systems have been proposed, for example, to track exceptions, state, probability, and I/O. There is a vast body of previous work. Various authors have considered their semantics, polymorphic effects, effect inference, effectful metalanguages, and other topics.

One area has been neglected. Previous work on effect systems has generally only considered call-by-value languages. Call-by-name and, in particular, call-by-need, have been ignored.

There is an obvious reason for this: call-by-need languages such as Haskell usually prefer to use monads to encapsulate effects. This is because it would otherwise be difficult for the programmer to understand the behaviour of programs. However, these languages are still not entirely pure. Many traditional examples

Dylan McDermott*, Computer Laboratory, University of Cambridge,
Dylan.McDermott@cl.cam.ac.uk
Alan Mycroft, Computer Laboratory, University of Cambridge,
Alan.Mycroft@cl.cam.ac.uk

of effects, such as nontermination and exceptions, are often included as impure features. Time and space usage and generative declarations can also be thought of as effects. Probabilistic behaviour does not suffer from the same problem with understanding program behaviour and therefore can reasonably be included in a lazy language. These are good reasons for determining how to track the effects of programs in call-by-need languages.

Tracking effects is a harder problem for call-by-need than for call-by-value and call-by-name. To see why, consider the following programs

$$\begin{array}{ll} \mathsf{let}\, x = \mathsf{inc}\,()\, \mathsf{in} & \mathsf{let}\, x = \mathsf{inc}\,()\, \mathsf{in} \\ \mathsf{let}\, y = \mathsf{inc}\,()\, \mathsf{in} & \mathsf{let}\, y = x\, \mathsf{in} \\ y + x & y + x \end{array}$$

Here, we assume that inc increments some integer state, and that $+$ evaluates its operands from left to right.

Which parts of the two example programs increment the state? For call-by-value this is easy to determine: it is incremented wherever we call inc. For call-by-name it is slightly less obvious, but we can see that in both programs x and y will call $\mathsf{inc}\,()$, so the state is incremented twice when we evaluate the addition. For call-by-need it is much harder to determine. In the program on the left, the state will be incremented twice: once when we use y and once when we use x. In the other program it will only be incremented once, and this will happen at the use of y. The use of x will not increment the state. In call-by-need, *using* one variable can change the *effect* of another.

The key difficulty in giving an effect system is determining where arguments and let-bound variables are evaluated. To do this we give a *coeffect* system [1]. Coeffects allow requirements programs have on their environments to be tracked. In this case, the requirements are the values of variables. As we show, we can determine where arguments are evaluated using coeffects for eager and lazy languages. This is the first step towards giving an effect system.

We then show that, with some small modifications, we can use the coeffect system to track effects. This gives a standalone system that tracks effects while tracking uses of variables, with the end result that we can determine the effects of programs.

Contributions We make the following contributions:

- we describe an effect system NAME for a call-by-name lambda-calculus (Section 2.3), extending the classical call-by-value system;
- we describe a coeffect system VARIABLE that determines when arguments are evaluated (Section 3). This coeffect system is parameterized by an al-

gebra that depends on the reduction strategy. We show that it can be instantiated for call-by-value and call-by-name;
– we show that VARIABLE can be instantiated for call-by-need (Section 4);
– we show how to refine VARIABLE to form a coeffect system EFFECT (Section 5). This gives a standalone system that can then be used to form an effect system. In particular, this gives us an effect system NEED for call-by-need;
– we add recursion to both VARIABLE and EFFECT (Section 6).

2 Effect systems

First we will describe *effect systems*. Each effect system is parameterized by a set $\mathcal{F}$. Elements $f \in \mathcal{F}$ are called *effects*. Following Katsumata [2], we assume that this set forms a preordered monoid $\langle \mathcal{F}, \leq, \bullet, 1 \rangle$. This means that $\langle \mathcal{F}, \bullet, 1 \rangle$ is a monoid, $\langle \mathcal{F}, \leq \rangle$ is a preorder and the binary operation $\bullet$ is monotone in both arguments. $\leq$ represents subeffecting, $\bullet$ sequencing of effects, and 1 the effect of a pure expression.

An effect system then consists of rules defining a judgment $\Gamma \vdash e : A \,\&\, f$, which means that in context Γ the expression e has type A and effect f. Effect systems are usually defined by a collection of inference rules, which give us a syntactic method of reasoning about programs with effects.

2.1 Examples

Traditional effect systems In traditional Gifford and Lucassen-style effect systems [3], there is some set Σ of *operations* (for example, $\{\mathsf{read}, \mathsf{write}\}$), and the preordered monoid is $\langle \mathcal{P}\,\Sigma, \subseteq, \cup, \emptyset \rangle$. An effect $f \subseteq \Sigma$ is the set of operations that an expression *may* perform. Subeffecting allows operations that are not performed to be added (for example to balance the effects of the two branches of an if), resulting in a *may* analysis. We can also consider variants of this, for example, if we switch from sets of operations to multisets of operations, we can count how many times each operation is used. The monoid $\langle \mathcal{P}\Sigma, \supseteq, \cup, \emptyset \rangle$ describes a *must* analysis.

Global state Consider a language that has a single global state of type S, and constants $\mathsf{read} : \mathsf{unit} \to S$ and $\mathsf{write} : S \to \mathsf{unit}$. In general, a program might need an initial state to run, but if we know that it must call write before it calls read, it does not. We can determine this information statically using an effect

system with the effects $\mathsf{writesFirst} \leq 1 \leq \mathsf{readsFirst}$. Here $\mathsf{writesFirst}$ means must write, and does not read before the first write; the unit 1 means does not read before the first write, but may do neither; and $\mathsf{readsFirst}$ means may read first. The sequencing operation is defined by

$$\begin{aligned} \mathsf{readsFirst} \bullet f &= \mathsf{readsFirst}\,; \\ 1 \bullet f &= f\,; \\ \mathsf{writesFirst} \bullet f &= \mathsf{writesFirst}\,. \end{aligned}$$

The effect of applying read is $\mathsf{readsFirst}$, and the effect of applying write is $\mathsf{writesFirst}$. Programs that have the effect $\mathsf{writesFirst}$ or the effect 1 do not need an initial state.

Nondeterminism Suppose that we have a language that contains a construct or, where $e_1 \,\mathsf{or}\, e_2$ means evaluate both e_1 and e_2 and then nondeterministically choose either as the result. For example, $(0 \,\mathsf{or}\, 1) + (1 \,\mathsf{or}\, 2)$ would evaluate to any element of the set $\{1, 2, 3\}$. We can use an effect system to place an upper bound on the number of results of an expression. The preordered monoid we use to do this consists of the positive integers with the usual ordering, multiplication for sequencing, and 1 for the effect of a pure expression

$$\langle \mathbb{N}_+, \leq, \cdot, 1 \rangle\,.$$

The typing rule for or would then be

$$\frac{\Gamma \vdash e_1 : A \,\&\, f_1 \qquad \Gamma \vdash e_2 : A \,\&\, f_2}{\Gamma \vdash e_1 \,\mathsf{or}\, e_2 : A \,\&\, f_1 + f_2}\,.$$

We would thus have:

$$\cdot \vdash (0 \,\mathsf{or}\, 1) + (1 \,\mathsf{or}\, 2) : \mathsf{int} \,\&\, 4\,.$$

In this example, the effect is overapproximated: the type system does not capture the fact that $1 + 1$ and $0 + 2$ both evaluate to 2, and therefore counts both separately.

External resource usage We can also use an effect system to count how many times an external resource is accessed. For example, we can use the preordered monoid $\langle \mathbb{N}, \leq, +, 0 \rangle$ to place an upper bound on the number of times a particular operation is performed. More complex type systems have also been used to place bounds on the amount of time and space programs use, for example, in Hoffmann et al.'s Resource Aware ML [4].

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \mid \\
 & & & c \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
\text{Types} & A, B & ::= & b \mid A \xrightarrow{f} B \\
\text{Contexts} & \Gamma & ::= & \overline{x : A}
\end{array}
$$

$$
(\text{var})\ \frac{}{\Gamma \vdash x : A \,\&\, 1}\ \text{if } (x : A) \in \Gamma \qquad (\text{sub})\ \frac{\Gamma \vdash e : A \,\&\, f}{\Gamma \vdash e : A \,\&\, f'}\ \text{if } f \leq f'
$$

$$
(\text{abs})\ \frac{\Gamma, x : A \vdash e : B \,\&\, f}{\Gamma \vdash \lambda x.\, e : A \xrightarrow{f} B \,\&\, 1}\ \text{if } x \notin \operatorname{dom} \Gamma
$$

$$
(\text{app})\ \frac{\Gamma \vdash e_1 : A \xrightarrow{f''} B \,\&\, f \qquad \Gamma \vdash e_2 : A \,\&\, f'}{\Gamma \vdash e_1\, e_2 : B \,\&\, f \bullet f' \bullet f''}
$$

$$
(\text{const})\ \frac{}{\Gamma \vdash c : A \,\&\, 1}\ \text{if } c : A \text{ is a constant}
$$

$$
(\text{if})\ \frac{\Gamma \vdash e_1 : \mathsf{bool} \,\&\, f \qquad \Gamma \vdash e_2 : A \,\&\, f' \qquad \Gamma \vdash e_3 : A \,\&\, f'}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A \,\&\, f \bullet f'}
$$

Fig. 1: Call-by-value effect system VALUE.

2.2 Call-by-value

An effect system VALUE for a call-by-value lambda calculus is given in Figure 1. We assume some collection of constants ranged over by c and some collection of base types ranged over by b. In particular, we assume that there are constants $\mathsf{true} : \mathsf{bool}$ and $\mathsf{false} : \mathsf{bool}$ (where bool is a base type). We could also have, for example, $\mathsf{read} : \mathsf{unit} \xrightarrow{\mathsf{readsFirst}} \mathsf{int}$.

Function types $A \xrightarrow{f} B$ are annotated with the *latent* effect of the function. Variables, constants, and lambda abstractions are pure; application evaluates the function, then the argument, and then the body of the function. The only non-syntax-directed rule is (sub), which allows effects to be overapproximated. The overapproximation is necessary for programs that use if. We do not attempt to determine which branch will be taken, so we require both branches to have the same effect.

$$\begin{array}{llll} \text{Expressions} & e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \mid \\ & & & c \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\ \text{Types} & A, B & ::= & b \mid A \xrightarrow{f', f''} B \\ \text{Contexts} & \Gamma & ::= & \overline{x : \langle A \rangle_f} \end{array}$$

$$\text{(var)}\ \frac{}{\Gamma \vdash x : A \,\&\, f}\ \text{if}\ (x : \langle A \rangle_f) \in \Gamma \qquad \text{(sub)}\ \frac{\Gamma \vdash e : A \,\&\, f}{\Gamma \vdash e : A \,\&\, f'}\ \text{if}\ f \leq f'$$

$$\text{(abs)}\ \frac{\Gamma, x : \langle A \rangle_{f'} \vdash e : B \,\&\, f''}{\Gamma \vdash \lambda x.\, e : A \xrightarrow{f', f''} B \,\&\, 1}\ \text{if}\ x \notin \operatorname{dom} \Gamma$$

$$\text{(app)}\ \frac{\Gamma \vdash e_1 : A \xrightarrow{f', f''} B \,\&\, f \qquad \Gamma \vdash e_2 : A \,\&\, f'}{\Gamma \vdash e_1\, e_2 : B \,\&\, f \bullet f''}$$

$$\text{(const)}\ \frac{}{\Gamma \vdash c : A \,\&\, 1}\ \text{if}\ c : A \text{ is a constant}$$

$$\text{(if)}\ \frac{\Gamma \vdash e_1 : \mathsf{bool} \,\&\, f \qquad \Gamma \vdash e_2 : A \,\&\, f' \qquad \Gamma \vdash e_3 : A \,\&\, f'}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A \,\&\, f \bullet f'}$$

Fig. 2: Call-by-name effect system NAME.

2.3 Call-by-name

We can give a similar effect system NAME for call-by-name (Figure 2). This is more difficult because uses of variables are no longer pure: they have whatever effect the corresponding argument has. We deal with impure variables by modifying typing contexts so that in addition to the type, they associate each variable with an effect. In a typing context, $x : \langle A \rangle_f$ means the variable x is a thunk that returns a value of type A and has effect f. Function types now have the form $A \xrightarrow{f', f''} B$, where f'' is the latent effect of the function and f' is the effect of the argument it accepts.

The (app) rule specifies the required effect of the argument. Note that the conclusion of the (app) rule does not mention the effect f' of the argument at all. This is because the latent effect f'' accounts for the effect of the argument. If the function uses its argument, the effect f' will in some sense be "included" in f'' because of a use of the (var) rule. Returning to the nondeterminism example,

we have

$$\cdot \vdash \lambda x.\, x + x : \mathsf{int} \xrightarrow{3,6} \mathsf{int} \,\&\, 1\,.$$

indicating that if we apply this function to an argument with three possible values, the result will have six possible values. Now applying the (app) rule we get

$$\cdot \vdash (\lambda x.\, x + x)\,(0 \,\mathsf{or}\, 1 \,\mathsf{or}\, 2) : \mathsf{int} \,\&\, 6\,.$$

We do not have to use the fact that the argument has three possible values in the conclusion of the (app) rule, because we already use it when typing the function.

Note that a practical type system would probably add effect polymorphism to this. In general if information is added to the typing context, polymorphism might be useful. The type system Variable, defined in Section 3, effectively has polymorphism built in since it avoids adding additional information to the typing context. In that section the coeffect substitution $\circledast_{\_}$ performs roughly the same task as a specialization rule in a polymorphic type system.

2.4 Call-by-need

Both of the above effect systems work by recording the effect of the argument of a function wherever it is evaluated. For both call-by-value and call-by-name, statically determining where each argument is evaluated is trivial. However, as we have already seen, it is harder for call-by-need. An argument is evaluated the first time the corresponding variable is used, but it is not obvious where this is. This is the primary difficulty in designing a call-by-need effect system. We therefore concentrate on solving this problem first.

3 Tracking variable usage

In this section we describe a system Variable that tracks when the argument corresponding to each variable is evaluated. That is, if we have an expression with free variables, it determines the (possibly multiple) orders in which the arguments corresponding to each variable will be evaluated.

Variable is parameterized by a *trace algebra* that describes how the reduction strategy behaves. The primary reason for developing it is for call-by-need, but we show that it is more general. We will first describe Variable in general and give the rules it contains, and then instantiate it for call-by-value, call-by-name and (most importantly) call-by-need.

Variable is a *coeffect system* [1]. That is, it consists of a set of inference rules defining a typing judgment of the form

$$\Gamma \mathbin{@} R \vdash e : A$$

where R ranges over *coeffects*. Coeffects are similar in many ways to effects, except that they describe what the program requires from the context, rather than how they affect it. In this case, R contains the information about variable usage that we wish to determine in order to give an effect system. The coeffect approximates when we *require* each variable from the context. We do not attempt to describe coeffects in general; we only show how to use them for variable tracking.

3.1 Traces

What exactly should the coeffect R tell us? First note that the set of variables the expression uses is not enough. To see why, consider the expressions

$$e = (\lambda y.\, (\lambda z.z + y)\, y)\, (\mathsf{read}\, ())$$

and

$$e' = (\lambda y.\, (\lambda z.y + z)\, y)\, (\mathsf{read}\, ())\,,$$

and suppose we evaluate them using call-by-need.

The expression e uses z before y, so z will have the effect readsFirst (since evaluating z will require getting the value of y, which has not been evaluated yet). However, e' evaluates y first, so z will be pure. We therefore need to track the orders in which variables are used to give an effect system. Hence we track *traces*: the variables that are evaluated, and the order in which they are evaluated. Coeffects are then sets of traces.

Traces u are given by the following grammar

$$u ::= \varepsilon \mid x \mid uu'\,.$$

A trace is either the empty trace ε (meaning no variables are used), a variable x (meaning that only x is used), or the concatenation uu' of two traces (meaning first the variables in u are used, and then the variables in u'). Concatenation is associative, and ε is the unit. This makes sense because we care about the order in which variables are used, not how uses are bracketed. These properties are used in the proof of type preservation for call-by-need (Theorem 4). The equivalence relation $\equiv$ (defined in Figure 3) makes them precise. (There is a

$$\frac{}{\varepsilon u \equiv u} \qquad \frac{}{u\varepsilon \equiv u} \qquad \frac{}{u_1(u_2u_3) \equiv (u_1u_2)u_3} \qquad \frac{u_1 \equiv u_1' \qquad u_2 \equiv u_2'}{u_1u_2 \equiv u_1'u_2'}$$

$$\frac{}{u \equiv u} \qquad \frac{u \equiv u'}{u' \equiv u} \qquad \frac{u \equiv u' \qquad u' \equiv u''}{u \equiv u''}$$

(a) Definition of $\equiv$

$$\begin{aligned} \mathrm{fv}\,\varepsilon &= \emptyset \\ \mathrm{fv}\,x &= \{x\} \\ \mathrm{fv}\,(uu') &= \mathrm{fv}\,u \cup \mathrm{fv}\,u' \end{aligned}$$

(b) Definition of fv on traces

$$\begin{aligned} \varepsilon[u/x] &= \varepsilon \\ x[u/x] &= u \\ y[u/x] &= y && \text{if } x \neq y \\ (u'u'')[u/x] &= (u'[u/x])(u''[u/x]) \end{aligned}$$

(c) Definition of trace substitution $u[u'/x]$

Fig. 3: Traces for VARIABLE.

reason for defining a separate equivalence relation instead of just describing traces as strings: when we consider effects again in Section 5, we will make $\equiv$ a little more complicated.)

From now on, we will identify traces up to $\equiv$. They are therefore just lists of variables (i.e. the free monoid on the set of variables).

We need some extra operations on traces. $\mathrm{fv}\,u$ is the set of variables that appear in the trace u, and $u[u'/x]$ is the substitution of the trace u' for x in u. The definitions of both are simple, and are given in Figure 3.

We can now say what the coeffect is in the judgment $\Gamma \mathbin{@} R \vdash e : A$. The coeffect R is a set of traces, each representing a possible behaviour of the program. When executing programs we can record where each argument is reduced, giving us an execution trace, and this will be (by type safety) one of the traces in R.

Coeffects are allowed to contain multiple traces so that variable usage can be approximated. We cannot expect to determine a precise effect for a program that uses a branching construct such as if, because that would require us to determine when the condition is true or false. To avoid having to do this, we assume that either branch of an if can be taken, which means we have to consider expressions as having several possible traces. The analysis will therefore be imprecise in the same way that the call-by-value effect system is imprecise for programs that use if.

3.2 Specifying a reduction strategy

Variable is parameterized by some additional data, which we call a *trace algebra*. We first give the definition, and then explain what each part of it means.

Definition 1. A *trace algebra* is a 4-tuple $\langle \oplus, \circledast\_, \mathrm{lat}\_, \mathrm{use}\_ \rangle$ consisting of a binary operation $\oplus$ on traces, and for each variable x, a binary operation $\circledast_x$ on traces, a function lat_x from traces to traces, and a constant trace use_x, such that the following conditions on free variables hold

$$\begin{aligned} \mathrm{fv}\,(u \oplus u') &\subseteq \mathrm{fv}\,u \cup \mathrm{fv}\,u'\,; \\ \mathrm{fv}\,(u \circledast_x u') &\subseteq (\mathrm{fv}\,u - \{x\}) \cup \mathrm{fv}\,u'\,; \\ \mathrm{fv}\,(\mathrm{lat}_x\,u) &\subseteq \{x\} \cup \mathrm{fv}\,u\,; \\ \mathrm{fv}\,\mathrm{use}_x &\subseteq \{x\}\,. \end{aligned}$$

The binary operation $\oplus$ is used for sequencing. If we reduce two expressions in sequence, giving the traces u and u', then $u \oplus u'$ gives the trace of the sequenced reduction. As we will see, this is not always concatenation of traces.

The binary operation $\circledast\_$ models substitution. If we have an expression that uses a variable x and has trace u (where u might mention x) and we substitute an expression with trace u' for x, then $u \circledast_x u'$ gives the trace of the result. When x appears in u it means we will evaluate the argument corresponding to x at that point. Hence we would expect occurrences of x to be replaced with the trace u'. However, $\circledast_x$ is not always substitution of traces.

The unary operation lat_x models variable binding. If an expression has a free variable x and a trace u (which might contain x), then $\mathrm{lat}_x\,u$ specifies how binding x affects u, giving the latent coeffect of a function. That is, it specifies the trace that the function has when it is applied. This is the *latent* trace of the function. (Compare this to the terminology used in effect systems: the latent effect of a function is the effect that a function will have once it is applied.)

Finally, the constant trace use_x specifies the trace of a use of x. Specifically, it specifies whether the argument corresponding to x will be evaluated when we use the variable x.

The conditions on free variables ensure that these operations only refer to variables that are in scope where we use them. For example, we use lat_x where x is bound, and therefore we allow it to contain x. More specifically, the conditions are used to prove Lemma 2. The first two free variable conditions are equalities in all of our examples, but the final two are not.

Examples We give the trace algebras for call-by-value and call-by-name as examples, deferring the discussion of call-by-need to Section 4.

For call-by-value, we use

$$\begin{aligned} u \oplus u' &= uu' \,; \\ u \circledast_x u' &= u[u'/x] \,; \\ \mathrm{lat}_x\, u &= xu \,; \\ \mathrm{use}_x &= \varepsilon \,. \end{aligned}$$

The trace $u \oplus u'$ of the sequencing of two expressions is just the concatenation of the two traces. If we substitute one expression into another then the resulting trace $u \circledast_x u'$ is just the substitution of the traces: if u says it will evaluate the expression we substitute in, then we will execute the entire trace u' at that point. The value of $\mathrm{lat}_x\, u$ states that a function first evaluates its argument before doing any other work by prepending x. Here we consider the argument to be evaluated as part of the function, but the call-by-value effect system (Figure 1) included the evaluation of the argument in the application rule. This is only a difference in presentation. Finally use_x is the empty trace: uses of variables do not do any additional evaluation in call-by-value.

For call-by-name, we use

$$\begin{aligned} u \oplus u' &= uu' \,; \\ u \circledast_x u' &= u[u'/x] \,; \\ \mathrm{lat}_x\, u &= u \,; \\ \mathrm{use}_x &= x \,. \end{aligned}$$

The traces $u \oplus u'$ and $u \circledast_x u'$ are the same as for call-by-value. However, $\mathrm{lat}_x u$ does not add x to the beginning of the trace: we do not eagerly evaluate the argument in call-by-name. The trace use_x is x here because whenever we use a variable we evaluate the expression again.

3.3 Coeffect system

We now describe the coeffect system Variable.

Most of the types in Variable are standard, but function types are slightly harder because they include latent information that may refer to their argument. They are written $(x : A) \xrightarrow{R} B$. The coeffect R is the set of traces of the body of the function (the latent information). This may need to refer to the argument of the function, and hence the function type *binds* the variable x, which can be used to refer to it. Similarly B may contain coeffects, and these may mention x. The type does not bind x inside A. The binding behaviour is therefore identical to that of dependent function types (which inspire the syntax), but the variable is only used in coeffects. We therefore do not have a full dependent type system.

As an example, elements of the type

$$(x : \mathsf{bool}) \xrightarrow{\{x\}} \mathsf{bool}$$

are functions that use their argument and no other variables. This could be any function on bool in call-by-value (since any function would evaluate its argument exactly once) or a function that uses its argument exactly once in call-by-name. In call-by-need, it would be a function that uses its variable at least once. Since types bind variables, we identify them up to α-conversion. Hence the above type is equal to

$$(y : \mathsf{bool}) \xrightarrow{\{y\}} \mathsf{bool}\,.$$

Since types may have free variables, we define the set $\mathrm{fv}\, A$ of free variables of the type A as follows

$$\mathrm{fv}\, b = \emptyset\,;$$
$$\mathrm{fv}\,((x : A) \xrightarrow{R} B) = \mathrm{fv}\, A \cup (\mathrm{fv}\, B - \{x\})\,.$$

We also assume that constants have types with no free variables.

Typing contexts Γ are ordered lists of variables with types. It is possible to formulate the coeffect system using unordered contexts, but the ordering allows a simpler statement of the soundness theorem we give in Section 4. We assume that in each typing context the types can only refer to variables that are introduced on the left, that is, in a typing context $\Gamma, x : A, \Gamma'$ we have $\mathrm{fv}\, A \subseteq \mathrm{dom}\,\Gamma$.

To give the rules of the coeffect system we need to lift some parts of the trace algebra. The operations $\oplus$, $\circledast\_$ and $\mathrm{lat}\_$ are lifted to coeffects elementwise

(recall that a coeffect R is a set of traces):

$$\begin{aligned} R \oplus R' &= \{u \oplus u' \mid u \in R \wedge u' \in R'\}\,; \\ R \circledast_x R' &= \{u \circledast_x u' \mid u \in R \wedge u' \in R'\}\,; \\ \mathrm{lat}_x\, R &= \{\mathrm{lat}_x\, u \mid u \in R\}\,. \end{aligned}$$

We do not need to lift the constant trace use_x.

We also lift $\circledast\_$ to types, so that if B is a type, x is a variable name and R is a coeffect then $B \circledast_x R$ is a type. The definition is capture-avoiding (and therefore α-conversion on types is important here)

$$b \circledast_x R' = b \qquad \text{for base types } b$$

$$((y : A) \xrightarrow{R} B) \circledast_x R' = (y : A \circledast_x R') \xrightarrow{R \circledast_x R'} (B \circledast_x R') \qquad \text{if } x \neq y$$

We will also need to consider the free variables of a coeffect:

$$\mathrm{fv}\, R = \bigcup_{u \in R} \mathrm{fv}\, u\,.$$

The typing judgement has the form $\Gamma \mathbin{@} R \vdash e : A$, and means the expression e has type A and coeffect R in context Γ. It is defined in Figure 4. Apart from (sub), the rules are syntax directed.

For variables x, the coeffect is $\{\mathrm{use}_x\}$. The subsumption rule (sub) allows additional traces to be added to the coeffect so that it can be overapproximated. The condition on free variables ensures that coeffects do not contain variables that are not in the typing context: the coeffect cannot say the expression uses a variable that is not in scope.

In a lambda abstraction, we use lat_x to determine the latent coeffect of the function. The immediate coeffect (the coeffect of the expression) is $\{\varepsilon\}$, meaning that evaluating a lambda abstraction does not immediately use a variable. Note that it might be interesting to consider reduction strategies that evaluate under lambdas. They could evaluate a part of the function that does not require the argument. The coeffect calculus presented in [5] allows situations like this, where the immediate coeffect is non-trivial. We do not pursue this here.

An application $e_1\, e_2$ first evaluates e_1 and then the body of the function, with e_2 substituted for the variable. Recall that in the call-by-value case, the latent coeffect takes care of the evaluation of e_2 at the beginning of the function. Since B might mention x, we also need to apply $\circledast_x$ to B.

The coeffect of an if is the sequencing of the coeffects of the condition and the (common) coeffect of the branches. The rule requires both branches to have the same coeffect; the subsumption rule allows this to be the case. We only need to

$$
\begin{array}{llll}
\text{Expressions} & e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \mid \\
 & & & c \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
\text{Types} & A, B & ::= & b \mid (x : A) \xrightarrow{R} B \\
\text{Contexts} & \Gamma & ::= & \overline{x : A}
\end{array}
$$

$$\text{(var)}\ \frac{}{\Gamma \mathbin{@} \{\mathrm{use}_x\} \vdash x : A}\ \text{if } (x : A) \in \Gamma$$

$$\text{(sub)}\ \frac{\Gamma \mathbin{@} R \vdash e : A}{\Gamma \mathbin{@} R' \vdash e : A}\ \text{if } R \subseteq R' \wedge \mathrm{fv}\, R' \subseteq \mathrm{dom}\, \Gamma$$

$$\text{(abs)}\ \frac{\Gamma, x : A \mathbin{@} R \vdash e : B}{\Gamma \mathbin{@} \{\varepsilon\} \vdash \lambda x.\, e : (x : A) \xrightarrow{\mathrm{lat}_x R} B}\ \text{if } x \notin \mathrm{dom}\, \Gamma$$

$$\text{(app)}\ \frac{\Gamma \mathbin{@} R \vdash e_1 : (x : A) \xrightarrow{R'} B \qquad \Gamma \mathbin{@} R'' \vdash e_2 : A}{\Gamma \mathbin{@} R \oplus (R' \circledast_x R'') \vdash e_1\, e_2 : B \circledast_x R''}$$

$$\text{(const)}\ \frac{}{\Gamma \mathbin{@} \{\varepsilon\} \vdash c : A}\ \text{if } c : A \text{ is a constant}$$

$$\text{(if)}\ \frac{\Gamma \mathbin{@} R \vdash e_1 : \mathsf{bool} \qquad \Gamma \mathbin{@} R' \vdash e_2 : A \qquad \Gamma \mathbin{@} R' \vdash e_3 : A}{\Gamma \mathbin{@} R \oplus R' \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A}$$

Fig. 4: Variable tracking coeffect system VARIABLE.

use (sub) when typing an if. As a result, coeffect inference is easy for VARIABLE: when inferring the coeffect of an if, use the (sub) rule to union the coeffects of the two branches; otherwise, just use the syntax-directed rules. Note that if we were to use the standard encoding of if using lambdas (so $\mathsf{if} = \lambda f.\lambda x.\lambda y.\, f\, x\, y$), we would get slightly different results to the rule (if). This is to be expected: for call-by-value, if we encode if as a function both branches would always be executed.

The following lemma states that coeffects can only refer to variables in the typing context. It uses the conditions on free variables in Definition 1.

Lemma 2. *If* $\Gamma \mathbin{@} R \vdash e : A$*, then* $\mathrm{fv}\, R \subseteq \mathrm{dom}\, \Gamma$.

Proof. We prove a slightly stronger property, namely that if $\Gamma \mathbin{@} R \vdash e : A$ then $\mathrm{fv}\, R \cup \mathrm{fv}\, A \subseteq \mathrm{dom}\, \Gamma$. The proof is by induction on the derivation of $\Gamma \mathbin{@} R \vdash e : A$.

- Case (var): By assumption $\operatorname{fv} \operatorname{use}_x \subseteq \{x\}$, so $\operatorname{fv} \{\operatorname{use}_x\} \subseteq \{x\} \subseteq \operatorname{dom} \Gamma$. Also recall that types in typing contexts types can only refer to variables that appear on the left, so we have $\operatorname{fv} A \subseteq \operatorname{dom} \Gamma$.
- Case (sub): From the side condition on the rule we have $\operatorname{fv} R' \subseteq \operatorname{dom} \Gamma$ and from the inductive hypothesis we have $\operatorname{fv} A \subseteq \operatorname{dom} \Gamma$.
- Case (abs): We have $\operatorname{fv} \{\varepsilon\} = \emptyset \subseteq \operatorname{dom} \Gamma$. By the inductive hypothesis we have $\operatorname{fv} R \subseteq \operatorname{dom} \Gamma \cup \{x\}$, so by the assumption on lat_x we also have $\operatorname{fv} (\operatorname{lat}_x R) \subseteq \operatorname{dom} \Gamma \cup \{x\}$. Hence (again using the rules on free variables in typing contexts)
$$\operatorname{fv} ((x : A) \xrightarrow{\operatorname{lat}_x R} B) \subseteq \operatorname{dom} \Gamma \,.$$
- Case (app): For the coeffect we have
$$\operatorname{fv} (R \oplus (R' \circledast_x R'')) \subseteq \operatorname{fv} R \cup ((\operatorname{fv} R' - \{x\}) \cup \operatorname{fv} R'') \,.$$
By the assumptions on $\oplus$ and $\circledast_x$,
$$\operatorname{fv} (R \oplus (R' \circledast_x R'')) \subseteq \operatorname{dom} \Gamma \,.$$
For the type we have
$$\operatorname{fv} (B \circledast_x R) \subseteq (\operatorname{fv} B - \{x\}) \cup \operatorname{fv} R \subseteq \operatorname{dom} \Gamma$$
from the assumption on $\circledast_x$.
- Case (const): We assumed that the type has no free variables, and the coeffect clearly has no free variables.
- Case (if): By the assumptions on $\oplus$, we have
$$\operatorname{fv} (R \oplus R') \subseteq \operatorname{fv} R \cup \operatorname{fv} R'$$
so by the inductive hypothesis, $\operatorname{fv} (R \oplus R') \subseteq \operatorname{dom} \Gamma$. For the type we have $\operatorname{fv} B \subseteq \operatorname{dom} \Gamma$ by the inductive hypothesis.

□

Expressions with multiple traces In general, an expression may have a coeffect that contains multiple traces. Each of these traces represents one possible behaviour. When reasoning about an expression, it is possible to reason about each possible behaviour individually, and then combine the information. Hence Variable helps us to analyze programs: we can analyse the program at each trace, using the information about variable usage that each trace provides, and then combine the results.

We will make use of this fact when giving the effect system in Section 5. When determining the effect of an expression we do it on individual traces and

combine the results. Since the effect of a variable in call-by-need depends on branches that were previously taken, attempting to describe the effect system directly would be significantly harder.

4 Call-by-need variable tracking

We now turn our attention to instantiating Variable for call-by-need. If we are using call-by-need, then only the first use of a variable will evaluate the corresponding argument. Hence traces should not repeat variables. If an expression uses the variable x twice, then x should appear only once in the trace. Otherwise, call-by-need behaves like call-by-name: arguments are evaluated where variables are used.

The function nub maps a trace to the trace in which only the first use of each variable is kept:

$$\begin{aligned} \operatorname{nub} \varepsilon &= \varepsilon\,; \\ \operatorname{nub}(ux) &= \operatorname{nub} u\,, && \text{if } x \in \operatorname{fv} u\,; \\ \operatorname{nub}(ux) &= (\operatorname{nub} u)x\,, && \text{if } x \notin \operatorname{fv} u\,. \end{aligned}$$

For example, $\operatorname{nub}(xyzx) = xyz$ and $\operatorname{nub}(xxyy) = xy$.

The trace algebra we use for call-by-need is simple to define. It is the same as for call-by-name except that nub retains only the first use of each variable in the trace:

$$\begin{aligned} u \oplus u' &= \operatorname{nub}(uu')\,; \\ u \circledast_x u' &= \operatorname{nub}(u[u'/x])\,; \\ \operatorname{lat}_x u &= u\,; \\ \operatorname{use}_x &= x\,. \end{aligned}$$

Variable can be instantiated with this algebra to give a coeffect system that tracks variable usage in call-by-need.

The resulting coeffect system has several interesting properties. One is that we never need to consider traces that repeat variables. Although it is possible to add traces with repeated variables using the (sub) rule, it is never necessary to do so. In particular, we have the following:

Lemma 3. *If we instantiate* Variable *for call-by-need then* $\Gamma \mathbin{@} R \vdash e : A$ *implies*

$$\Gamma \mathbin{@} \{u \in R \mid \operatorname{nub} u = u\} \vdash e : A\,.$$

Proof. The proof is by induction on the derivation of $\Gamma \mathbin{@} R \vdash e : A$. The rules (var), (abs) and (const) are trivial. For (app) and (if), the result follows from the fact that nub is applied to each trace in the coeffect and $\mathrm{nub} \circ \mathrm{nub} = \mathrm{nub}$. For (sub) the result is an immediate consequence of the inductive hypothesis. □

A useful consequence of this lemma is that we only need to consider finite coeffects: since each trace may only mention variables in $\mathrm{dom}\,\Gamma$ (Lemma 2), we do not need to consider traces that are longer than the number of variables in $\mathrm{dom}\,\Gamma$. There are only finitely many such traces. This fact should make it easier to use Variable for call-by-need in practice (though determining to what extent is out of scope of the present paper).

4.1 Soundness

We prove the soundness of the coeffect system for call-by-need relative to a small-step operational semantics, based on the semantics given by Launchbury [6]. We choose a heap-based semantics rather than a heapless one because it makes uses of variables more explicit: when an expression uses a variable for the first time, this is reflected by a change in the heap. The heap-based semantics also has the advantages that it is simple, and that it do not use syntactic substitution for expressions. It would likely be possible to prove a similar theorem for a heapless semantics, such as the one given by Ariola et al. [7].

A heap ρ is an ordered list of pairs. Each of the pairs in the heap either has the form $x \mapsto \mathsf{val}\, v$ or $x \mapsto \mathsf{expr}\, e$. The former case means that x has been evaluated, and its value is v. We use the following class of values:

$$v ::= c \mid \lambda x.\, e\,.$$

The latter case $x \mapsto \mathsf{expr}\, e$ means that x is the unevaluated expression e. To model laziness, when a variable is added to the heap it is initially an unevaluated expression, and the first time it is used it is replaced with its value.

Concatenation of heaps is written using a comma. Hence $\rho_1, x \mapsto \mathsf{expr}\, e, \rho_2$ is the concatenation of three heaps, where ρ_1 and ρ_2 are possibly empty. In this case the free variables of e will all be in ρ_1. In general, if a variable is free in some expression inside the heap, it will be bound somewhere to the left. Reductions preserve this property, and the typing judgement $\vdash_{\mathsf{h}}$ for heaps (defined below) enforces it. Note in particular that the comma is not commutative: the order in which variables are listed in the heap matters.

The judgement form for reduction is $e \mid \rho \overset{u}{\rightsquigarrow} e' \mid \rho'$. It is defined by the rules given in Figure 5. The trace u gives the variables that were evaluated for the

first time. Since this is a small-step semantics, u always consists of zero or one variables.

Reductions can only add variables to the right of the heap, and can never remove them. It is possible to consider a garbage collection rule that removes variables when they go out of scope, but this is unnecessary here.

We need only consider heaps in which every expression has a valid type. An augmented typing judgement $\vdash_{\mathsf{h}}$ for heaps captures this. If Γ is a context, Ψ is a list of coeffects of the same length as Γ, and ρ is a heap, then $\Gamma \mathbin{@} \Psi \vdash_{\mathsf{h}} \rho$ means ρ maps each variable in Γ to an expression or value with the type given in Γ and coeffect given in Ψ. It is defined inductively by the following rules:

$$\text{(empty)}\ \frac{}{\cdot \mathbin{@} \cdot \ \vdash_{\mathsf{h}}\ \cdot}\,; \qquad \text{(expr)}\ \frac{\Gamma \mathbin{@} \Psi \ \vdash_{\mathsf{h}}\ \rho \qquad \Gamma \mathbin{@} R \vdash e : A}{\Gamma, x : A \mathbin{@} \Psi, R \ \vdash_{\mathsf{h}}\ \rho, x \mapsto \mathsf{expr}\, e}\,;$$

$$\text{(val)}\ \frac{\Gamma \mathbin{@} \Psi \ \vdash_{\mathsf{h}}\ \rho \qquad \Gamma \mathbin{@} \{\varepsilon\} \vdash v : A}{\Gamma, x : A \mathbin{@} \Psi, \{\varepsilon\} \ \vdash_{\mathsf{h}}\ \rho, x \mapsto \mathsf{val}\, v}\,.$$

The rule (empty) states that the empty heap is well-typed in the empty context with the empty list of coeffects. The rule (expr) states that if the last variable in the heap maps to an expression, then that expression must have the correct type and coeffect, and the remainder of the heap must be well-typed. The (val) rule does the same when the last variable maps to a value. It also requires the coeffect to be $\{\varepsilon\}$.

Soundness should imply that if an expression reduces, producing a trace, then that trace should be in the coeffect of the expression. Heaps add some complexity here. Consider the expression e that is just the variable x. The expression e has coeffect $\{x\}$, but if e is reduced in the heap $y \mapsto \mathsf{expr}\,\mathsf{true}, x \mapsto \mathsf{expr}\, y$, the trace will be yx. The reduction also evaluates y, which does not appear in e. Variables may refer to suspended computations, which may evaluate other variables. This fact is accounted for by substituting the coeffects of the expressions in the heap into the coeffect of the expression being reduced. The order of the substitutions is important because expressions in the heap may refer to other expressions in the heap. Define coeffect substitution $R[\Psi/\rho]$ by

$$\begin{aligned} R[\cdot/\cdot] &= R\,; \\ R[(\Psi, R')/(\rho, x \mapsto \mathsf{expr}\, e)] &= (R \circledast_x (R' \oplus \{x\}))[\Psi/\rho]\,; \\ R[(\Psi, R')/(\rho, x \mapsto \mathsf{val}\, e)] &= (R \circledast_x \{\varepsilon\})[\Psi/\rho]\,. \end{aligned}$$

Here, we are using the trace algebra for call-by-need, so $\circledast\_$ is substitution with duplicates removed and $\oplus$ is concatenation with duplicates removed. The expr

$$(\text{var1})\ \frac{}{x \mid \rho_1, x \mapsto \mathsf{expr}\, v, \rho_2 \overset{x}{\rightsquigarrow} v \mid \rho_1, x \mapsto \mathsf{val}\, v, \rho_2}$$

$$(\text{var2})\ \frac{}{x \mid \rho_1, x \mapsto \mathsf{val}\, v, \rho_2 \overset{\varepsilon}{\rightsquigarrow} v \mid \rho_1, x \mapsto \mathsf{val}\, v, \rho_2}$$

$$(\text{var3})\ \frac{e \mid \rho_1 \overset{u}{\rightsquigarrow} e' \mid \rho_1'}{x \mid \rho_1, x \mapsto \mathsf{expr}\, e, \rho_2 \overset{u}{\rightsquigarrow} x \mid \rho_1', x \mapsto \mathsf{expr}\, e', \rho_2}$$

(a) Variable rules

$$(\beta)\ \frac{}{(\lambda x.e_1)e_2 \mid \rho \overset{\varepsilon}{\rightsquigarrow} e_1 \mid \rho, x \mapsto \mathsf{expr}\, e_2}\ \text{if } x \notin \operatorname{dom} \rho$$

$$(\text{app})\ \frac{e_1 \mid \rho \overset{u}{\rightsquigarrow} e_1' \mid \rho'}{e_1\, e_2 \mid \rho \overset{u}{\rightsquigarrow} e_1'\, e_2 \mid \rho'}$$

(b) Application rules

$$(\text{if-true})\ \frac{}{\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 \mid \rho \overset{\varepsilon}{\rightsquigarrow} e_2 \mid \rho}$$

$$(\text{if-false})\ \frac{}{\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 \mid \rho \overset{\varepsilon}{\rightsquigarrow} e_3 \mid \rho}$$

$$(\text{if-cong})\ \frac{e_1 \mid \rho \overset{u}{\rightsquigarrow} e_1' \mid \rho'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \rho \overset{u}{\rightsquigarrow} \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \rho'}$$

(c) if rules

Fig. 5: Call-by-need operational semantics.

and val cases differ because if a variable has already been used further uses of it are not recorded.

Note that in traces, a variable appears when its evaluation ends (where expr is replaced with val in the heap). This choice is arbitrary. We could change the definition of the reduction relation (in particular the rules (var1), (var2) and (var3)) so that variables appear where their evaluation begins instead. This would then require a corresponding change to the definition of substitution of heaps for type preservation to hold.

Similarly, define type substitution $A[\Psi/\rho]$ by

$$b[\Psi/\rho] = b\,, \qquad \text{for base types } b\,;$$
$$((y : A) \xrightarrow{R} B)[\Psi/\rho] = (y : A[\Psi/\rho]) \xrightarrow{R[\Psi/\rho]} (B[\Psi/\rho])\,, \qquad \text{if } y \notin \operatorname{dom}\rho\,.$$

It is now possible to define a typing judgement $\vdash_{\mathsf{a}}$ that accounts for the computations in the heap. Write $\Gamma \mid \rho \mathbin{@} R \vdash_{\mathsf{a}} e : A$ if and only if there is some list of coeffects Ψ, type A' and coeffect R' such that

- $\Gamma \mathbin{@} \Psi \vdash_{\mathsf{h}} \rho\,;$
- $\Gamma \mathbin{@} R' \vdash e : A'\,;$
- $R = R'[\Psi/\rho]\,;$
- $A = A'[\Psi/\rho]\,.$

For example, recall that if we evaluate the expression x with the heap $y \mapsto \mathsf{expr}\,\mathsf{true}, x \mapsto \mathsf{expr}\,y$ then we get the trace yx. We have $y : \mathsf{bool}, x : \mathsf{bool} \mathbin{@} \{x\} \vdash x : \mathsf{bool}$, which implies

$$y : \mathsf{bool}, x : \mathsf{bool} \mid y \mapsto \mathsf{expr}\,\mathsf{true}, x \mapsto \mathsf{expr}\,y \mathbin{@} \{yx\} \vdash_{\mathsf{a}} x : \mathsf{bool}\,.$$

The coeffect therefore matches the trace.

Finally, note that variables that are bound by a lambda inside an expression are not mentioned in the coeffect, but reduction may add them to the heap. We therefore restrict the type and coeffect of the reduced expression to only the variables that were free in the original expression. (We write $R|_X$ for the restriction of the coeffect R to the variables in X, and similarly write $A|_X$ for the restriction of a type.)

The preservation theorem states that reduction preserves types, and that the change in the coeffect of the expression is given by the trace. Hence the coeffect system correctly tracks uses of variables.

Theorem 4 (Type preservation). *If* $\Gamma \mid \rho \mathbin{@} R \vdash_{\mathsf{a}} e : A$ *and* $e \mid \rho \overset{u}{\rightsquigarrow} e' \mid \rho'$ *then there is some context* Γ' *with* $\operatorname{dom}\Gamma' \supseteq \operatorname{dom}\Gamma$*, coeffect* R' *and type* A' *such that*

- $\Gamma' \mid \rho' \mathbin{@} R' \vdash_{\mathsf{a}} e' : A'\,;$

- $R \supseteq (\{u\} \oplus R')|_{\mathrm{dom}\,\Gamma}$;
- $A = A'|_{\mathrm{dom}\,\Gamma}$.

The proof is not difficult, but is tedious. Due to its length, we do not present the whole proof, but merely highlight some key parts of it.

Proof sketch. The proof is by induction on the derivation of $e \,|\, \rho \overset{u}{\rightsquigarrow} e' \,|\, \rho'$. For the rule (var1) we note that if the value v is typable with any coeffect at all then it must be typable with coeffect $\{\varepsilon\}$. We also note that if the variable x is typable with coeffect R, then $\{x\} \subseteq R$. Finally we use the fact that heap substitution, when x maps to some expr, leaves the occurrences of x in the coeffect, so that when we append the trace x to the coeffect we get the same coeffect as the result of the substitution.

For the rule (var2) we similarly use that values have coeffect $\{\varepsilon\}$.

For the congruence rules we have to consider subtyping. For example, for (app) we use the fact that if we have

$$\Gamma \mathbin{@} R \vdash e_1\, e_2 : B$$

then there exist types A and B' and coeffects S_1, S_2 and S_3 such that the following hold

- $S_1 \oplus (S_2 \circledast_x S_3) \subseteq R$;
- $B' \circledast_x S_3 = B$;
- $\Gamma \mathbin{@} S_1 \vdash e_1 : (x : A) \xrightarrow{S_2} B'$;
- $\Gamma \mathbin{@} S_3 \vdash e_2 : A$.

We use a similar fact for (if-cong).

The beta rules have no particular difficulties. □

Type preservation for other reduction strategies We do not prove a type preservation theorem for other reduction strategies, because we concentrate on call-by-need here. However, we briefly consider what we would need to prove one. The only changes we need to state a type preservation property for another reduction strategy are to replace the trace algebra, and to replace the operational semantics (the definition of $e|\rho \rightsquigarrow e'|\rho'$) with the correct (heap-based) operational semantics for the other reduction strategy. For call-by-name we can do this by deleting (var1) and (var3), and replacing (var2) with the following rule

$$\frac{}{x \,|\, \rho_1, x \mapsto \mathsf{expr}\, e, \rho_2 \overset{x}{\rightsquigarrow} e \,|\, \rho_1, x \mapsto \mathsf{expr}\, e, \rho_2} .$$

That is, we do not reduce expressions in the heap, but instead just copy entire expressions whenever the variable is used. Similarly, for call-by-value, we can

change the rules (var1) and (var3) with ones that allow use to reduce inside the heap at any time (not just when we are reducing a variable), and then change the application and if rules so that they require the heap to not contain expr, so that we are *forced* to reduce an argument when it is added to the heap. Of course, whether type preservation holds for a reduction strategy depends on the relationship between the operational semantics and the trace algebra. We do not yet have a way of proving a general type preservation property, that is, a type preservation property that is parameterized by both the trace algebra and the operational semantics.

5 Effect system

We have described how to determine when arguments are evaluated using a coeffect system Variable. This is the key difficulty in describing an effect system. We will now show that we can modify this system to get one that gives the effects of expressions. We will call this Effect. Effect is a refinement of Variable, and can be used as a standalone system to analyze both effects and variable usage.

Throughout this section, we assume that the algebra of effects is specified by a preordered monoid $\langle \mathcal{F}, \leq, \bullet, 1 \rangle$, with $\mathcal{F}$ and the set of variable names disjoint.

The first modification we make is to allow traces to contain effects $f \in \mathcal{F}$ as well as variables x. To do this, we replace the grammar of traces with

$$u ::= f \mid x \mid uu' .$$

Now we can have traces such as xfy, which means evaluate the argument corresponding to x, do some computation that has the effect f, and then evaluate the argument corresponding to y. The unit 1 of the effect algebra replaces ε: both mean there is no interaction with the environment that we care about.

Again traces are just strings, so concatenation is associative and has 1 as the unit element. However, we also allow effects that appear side-by-side in the string to be merged using $\bullet$, so if $f_1 \bullet f_2 = f_3$ then xf_1f_2 is the same as xf_3, but f_1xf_2 is not the same as xf_3 or f_3x. We again define an equivalence relation $\equiv$ to make this precise (Figure 6), and identify traces up to $\equiv$.

The set of variables $\mathrm{fv}\, u$ in a trace and substitution of traces are defined as before, but with $\mathrm{fv}\, f = \emptyset$ and $f[u/x] = f$ for $f \in \mathcal{F}$. Similarly for the function nub: we remove repeated occurrences of variables, but keep occurrences of effects.

The next change we make concerns the subsumption rule. Before there was no notion of an individual trace being more precise than another. x is not more

$$\frac{}{1u \equiv u} \qquad \frac{}{u1 \equiv u} \qquad \frac{}{ff' \equiv f \bullet f'} \qquad \frac{}{u_1(u_2u_3) \equiv (u_1u_2)u_3}$$

$$\frac{u_1 \equiv u_1' \qquad u_2 \equiv u_2'}{u_1u_2 \equiv u_1'u_2'} \qquad \frac{}{u \equiv u} \qquad \frac{u \equiv u'}{u' \equiv u} \qquad \frac{u \equiv u' \qquad u' \equiv u''}{u \equiv u''}$$

(a) Definition of $\equiv$

$$\begin{aligned} \mathrm{fv}\, f &= \emptyset \\ \mathrm{fv}\, x &= \{x\} \\ \mathrm{fv}\,(uu') &= \mathrm{fv}\, u \cup \mathrm{fv}\, u' \end{aligned}$$

(b) Definition of fv on traces

$$\begin{aligned} f[u/x] &= f \\ x[u/x] &= u \\ y[u/x] &= y \\ (u'u'')[u/x] &= (u'[u/x])(u''[u/x]) \end{aligned}$$

(c) Definition of trace substitution $u[u'/x]$

$$\begin{aligned} \mathrm{nub}\, f &= f \\ \mathrm{nub}\,(ux) &= \mathrm{nub}\, u && \text{if } x \in \mathrm{fv}\, u \\ \mathrm{nub}\,(ux) &= (\mathrm{nub}\, u)x && \text{if } x \notin \mathrm{fv}\, u \\ \mathrm{nub}\,(uf) &= (\mathrm{nub}\, u)f \end{aligned}$$

(d) Definition of nub

$$\frac{f \leq f'}{f \sqsubseteq f'} \qquad \frac{u_1 \sqsubseteq u_1' \qquad u_2 \sqsubseteq u_2'}{u_1u_2 \sqsubseteq u_1'u_2'} \qquad \frac{u \equiv u'}{u \sqsubseteq u'} \qquad \frac{u \sqsubseteq u' \qquad u' \sqsubseteq u''}{u \sqsubseteq u''}$$

(e) Definition of $\sqsubseteq$

Fig. 6: Traces for Effect.

precise than xy, it is just different. Since effects do have a notion of more precise (the order $\leq$), we extend it to traces, defining a relation $\sqsubseteq$. For example, we have that if $f \leq f'$ then $xf \sqsubseteq xf'$. Two traces are related by $\sqsubseteq$ only if they contain exactly the same variables in the same order, so we still do not have $x \sqsubseteq xy$ or $xy \sqsubseteq x$.

Using the order $\sqsubseteq$, traces form a preordered monoid. In fact, this is just the *free product* of the preordered monoid of Variable traces (the free monoid on the set of variables, using equality as the order) with the preordered monoid $\mathcal{F}$ of effects.

Again we parameterize the judgement by a trace algebra that depends on the reduction strategy. The trace algebras for Effect are identical to those for Variable except that they are defined on traces that can contain effects. Note that since all of the operations we used to specify trace algebras (such as concatenation and substitution) are also defined on the new traces, we can use the same algebras, but with ε replaced by 1. In fact, we *will* use the same ones: the algebra we use for call-by-value, call-by-name and call-by-need is exactly the same as before. We do not have to do any additional work to instantiate Effect than we did for Variable.

Again coeffects R are just sets of traces. We lift the preorder $\sqsubseteq$ on traces to a preorder on coeffects

$$R \sqsubseteq R' \quad \text{iff} \quad \forall u \in R.\, \exists u' \in R'.\, u \sqsubseteq u' \,.$$

In types A the only change we make is using this new kind of coeffect (in which the traces can contain effects). In a function type

$$(x : A) \xrightarrow{R} B \,,$$

R now provides the latent effect as well as the latent variable usage information. For example, if $R = \{f\}$ then the function will have the effect f when it is applied (and will not evaluate any arguments). The set R might contain multiple effects. In this case, it means the function will have at least one of the effects in R.

Effect consists of a judgement $\Gamma \mathbin{@} R \vdash e : A$. The inference rules are almost identical to the rules for Variable. The only changes are that ε is replaced with 1 and in the subsumption rule $\subseteq$ is replaced with $\sqsubseteq$. They are given in Figure 7.

Note that Effect subsumes Variable: if we use the trivial preordered monoid with underlying set $\mathcal{F} = \{\varepsilon\}$ as the effect algebra then the two are identical.

$$
\begin{array}{lrcl}
\text{Expressions} & e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \mid \\
 & & & c \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
\text{Types} & A, B & ::= & b \mid (x : A) \xrightarrow{R} B \\
\text{Contexts} & \Gamma & ::= & \overline{x : A}
\end{array}
$$

$$\text{(var)}\ \frac{}{\Gamma \mathbin{@} \{\mathrm{use}_x\} \vdash x : A}\ \text{if } (x : A) \in \Gamma$$

$$\text{(sub)}\ \frac{\Gamma \mathbin{@} R \vdash e : A}{\Gamma \mathbin{@} R' \vdash e : A}\ \text{if } R \sqsubseteq R' \wedge \mathrm{fv}\, R' \subseteq \mathrm{dom}\, \Gamma$$

$$\text{(abs)}\ \frac{\Gamma, x : A \mathbin{@} R \vdash e : B}{\Gamma \mathbin{@} \{1\} \vdash \lambda x.\, e : (x : A) \xrightarrow{\mathrm{lat}_x R} B}\ \text{if } x \notin \mathrm{dom}\, \Gamma$$

$$\text{(app)}\ \frac{\Gamma \mathbin{@} R \vdash e_1 : (x : A) \xrightarrow{R'} B \qquad \Gamma \mathbin{@} R'' \vdash e_2 : A}{\Gamma \mathbin{@} R \oplus (R' \circledast_x R'') \vdash e_1\, e_2 : B \circledast_x R''}$$

$$\text{(const)}\ \frac{}{\Gamma \mathbin{@} \{1\} \vdash c : A}\ \text{if } c : A \text{ is a constant}$$

$$\text{(if)}\ \frac{\Gamma \mathbin{@} R \vdash e_1 : \mathsf{bool} \qquad \Gamma \mathbin{@} R' \vdash e_2 : A \qquad \Gamma \mathbin{@} R' \vdash e_3 : A}{\Gamma \mathbin{@} R \oplus R' \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A}$$

Fig. 7: Inference rules for the coeffect system Effect.

5.1 Defining an effect system

How does this tell us the effect of an expression? Suppose that we have a closed expression e which is typable as $\cdot \mathbin{@} R \vdash e : A$, where $\cdot$ is the empty typing context. We saw earlier that for Variable, the free variables of the coeffect are a subset of the domain of the context (Lemma 2). The same is true for Effect, so in this case $\mathrm{fv}\, R = \emptyset$. Hence R is just a set of effects (recall that we identify traces up to $\equiv$). If evaluating e has effect $f \in \mathcal{F}$ then $f \in R$.

If we wish to assign a single effect to e then we can take the upper bound of R if it exists. The existence of upper bounds is not an issue specific to Effect: the usual effect system for call-by-value (Figure 1) has the same issue when typing an if.

We can define an effect system of the same form as in Section 2. As for NAME (Figure 2), typing contexts contain effects (we support both call-by-name and call-by-need so uses of variables might have effects). Now write

$$\overline{x : \langle A \rangle_f} \vdash e : A \,\&\, f'$$

if and only if there exists a coeffect R and type A' such that
- $\overline{x : A} \mathbin{@} R \vdash e : A$;
- $R \circledast_{x_1} \{f_1\} \dots \circledast_{x_n} \{f_n\} \sqsubseteq \{f'\}$;
- $A' = A \circledast_{x_1} \{f_1\} \dots \circledast_{x_n} \{f_n\}$.

In particular, if we use the trace algebra for call-by-need, this defines a call-by-need effect system NEED. If the effect monoid has all upper bounds, then if $\overline{x : A} \mathbin{@} R \vdash e : A$ then for all effects $\overline{f}$ there is always some effect f' we can assign to e If the effect monoid does not have all upper bounds then this might not be possible.

A notable feature of the effect system is that it supports effect polymorphism. That is, the same function can be used on arguments with different effects. This is not the case for the call-by-name effect system in Section 2.3.

5.2 Examples

We give some examples using the trace algebra for call-by-need that we described in Section 4 and using the effect algebra for nondeterminism that we described in Section 2.1, i.e. the preordered monoid

$$\langle \mathbb{N}_+, \leq, \cdot, 1 \rangle .$$

We can derive the following in EFFECT

$$\cdot \mathbin{@} \{1\} \vdash \lambda x.\, 1 + 1 : (x : \mathsf{int}) \xrightarrow{\{1\}} \mathsf{int};$$
$$\cdot \mathbin{@} \{1\} \vdash \lambda x.\, x + 1 : (x : \mathsf{int}) \xrightarrow{\{x\}} \mathsf{int};$$
$$\cdot \mathbin{@} \{1\} \vdash \lambda x.\, x + x : (x : \mathsf{int}) \xrightarrow{\{x\}} \mathsf{int}.$$

For the first expression, the function does not use its argument, and hence has latent coeffect $\{1\}$. The second evaluates its argument once, so has latent coeffect $\{x\}$. The third mentions the argument twice, but since we are in call-by-need, evaluates it once. Hence it *also* has latent coeffect $\{x\}$.

Now applying each of these expressions to the argument $0 \,\mathsf{or}\, 1$ (which has coeffect $\{2\}$) gives us

$$\cdot \mathbin{@} \{1\} \vdash (\lambda x.\, 1 + 1)\,(0 \,\mathsf{or}\, 1) : \mathsf{int}\,;$$
$$\cdot \mathbin{@} \{2\} \vdash (\lambda x.\, x + 1)\,(0 \,\mathsf{or}\, 1) : \mathsf{int}\,;$$
$$\cdot \mathbin{@} \{2\} \vdash (\lambda x.\, x + x)\,(0 \,\mathsf{or}\, 1) : \mathsf{int}\,.$$

We can now see that the first expression has effect 1, meaning that there is only one possible result:

$$\cdot \vdash (\lambda x.\, 1 + 1)\,(0 \,\mathsf{or}\, 1) : \mathsf{int} \,\&\, 1\,.$$

The second and third both have effect 2 (meaning that there are two possible results). Note that this relies critically on the fact that we use call-by-need: Value assigns the first expression the effect 2 and Name assigns the third the effect 4.

We can also use the effect system on expressions that involve if. In Effect we can derive

$$x : \mathsf{bool}, y : \mathsf{int} \mathbin{@} \{x, xy\} \vdash \mathsf{if}\ x\ \mathsf{then}\ 0\ \mathsf{else}\ y : \mathsf{int}$$

and hence

$$x : \langle \mathsf{bool} \rangle_1, y : \langle \mathsf{int} \rangle_2 \vdash \mathsf{if}\ x\ \mathsf{then}\ 0\ \mathsf{else}\ y : \mathsf{int} \,\&\, 2$$

since $\{1, 2\} \sqsubseteq \{2\}$. That is, if x chooses from only one value, and y chooses from two, then the if chooses from up to two values.

6 Recursion

We next consider how to support recursion. To do this, assume that the grammar of expressions is extended with fixed points as follows

$$e ::= \ldots \mid \mathsf{fix}\, e\,.$$

The expression $\mathsf{fix}\, e$ evaluates the fixed point of the function e. In this section we will not consider call-by-value, which usually only supports fixed points of the form $\mathsf{fix}\, v$ where v ranges over values, and concentrate on call-by-name and call-by-need.

Operationally $\mathsf{fix}\, e$ is equivalent to $e\,(\mathsf{fix}\, e)$. For call-by-need, we can extend the operational semantics given in Section 4.1 to support $\mathsf{fix}\, e$ by adding the following rule

$$\frac{}{\mathsf{fix}\, e \mid \rho \overset{\varepsilon}{\rightsquigarrow} e\,(\mathsf{fix}\, e) \mid \rho}\,.$$

Since $\mathsf{fix}\, e$ is equivalent to $e\,(\mathsf{fix}\, e)$, we should have that the coeffect R'' of $\mathsf{fix}\, e$ is a superset of the coeffect of $e\,(\mathsf{fix}\, e)$. So if e has immediate coeffect R and latent coeffect R', we would need to have

$$R'' \supseteq R \oplus (R' \circledast_x R'')\,.$$

We add the following rule (we use the same rule for both Variable and Effect):

$$(\text{fix})\ \frac{\Gamma \mathbin{@} R \vdash e : (x : A \circledast_x R) \xrightarrow{R'} A}{\Gamma \mathbin{@} R'' \vdash \mathsf{fix}\, e : A[R/x]} \text{ if } R'' \supseteq R \oplus (R' \circledast_x R'')\,.$$

Clearly, it is important to be able to solve the side condition in the rule to find R''. Perhaps surprisingly, a solution always exists. We can always take

$$R'' = \bigcup_{i \in \mathbb{N}} S_i$$

where S is the following sequence

$$\begin{aligned} S_0 &= \emptyset\,; \\ S_{i+1} &= R \oplus (R' \circledast_x S_i)\,. \end{aligned}$$

If we have a trace

$$u \oplus (u' \circledast_x u'') \in R \oplus (R' \circledast_x R'')$$

then we have $u'' \in S_i$ for some i, and hence

$$u \oplus (u' \circledast_x u'') \in S_{i+1} \subseteq R''$$

so R'' is a solution to the side condition.

This might be surprising because in general the effect algebra might not have a fixed point operation. For example, consider nondeterminism using the preordered monoid $\langle \mathbb{N}_+, \leq, \cdot, 1 \rangle$ as in Section 2. Using fix, we can write a program that makes an unbounded number of choices with or. Usually we have to add ∞ to give such a program an effect. However in this case (assuming the program uses no free variables), we can use the coeffect $R'' = \mathbb{N}_+$. If we added ∞, we would have $\mathbb{N}_+ \sqsubseteq \{\infty\}$.

The solution R'' may be unsatisfactory for two reasons. First, it might be infinite. This is the case for some effects, and also for Variable for call-by-name. The coeffect might also overapproximate.

If we consider only Variable (so traces do not include effects) then recall that we only need to consider finite coeffects for call-by-need. Hence this iterative construction terminates after a finite number of steps, and R'' is finite. This is also the most precise solution, and hence we do not have the same overapproximation. This solution would not necessarily work for Effect, because it depends on the effect algebra.

7 Related work

Effect systems As we noted in the introduction, a large amount of work has been done on effect systems, but primarily for call-by-value. Set-based call-by-value effect systems [3] are well-known. Katsumata [2] describes the more general form (with preordered monoids) we use here. It is also possible to consider effect systems with more operations to support other language constructs, such as concurrency [8]. It is likely that we can do the same for our system, by defining the same operations for traces. Other extensions to effect systems, such as regions [9] would also be interesting to consider. It has also been shown that effect systems are related to session types [10]. It may therefore be possible to apply our work to session types as well.

Coeffect systems Coeffect systems were first described in Petricek et al. [1], and were generalized by the same authors in [5]. They have previously been applied to, for example, implicit parameters, dataflow programming and liveness analysis. The coeffect system presented here has not been considered before. Coeffect systems have also been combined with effect systems by Gaboardi et al. [11]. Their system has more limited interaction between coeffects and effects than we require here.

Effects and call-by-name Some work has been done on denotational semantics for call-by-name languages using monads. Moggi [12] first mentions such a semantics, where any expression can have an effect. Benton et al. [13] give a slightly different one in which effects can only occur at base types.

None of this work considers effect *systems* for call-by-name. The natural denotational semantics for the effect system Name would be a modification of Moggi's translation to use *graded monads* [2]. An effect system that is closer to the translation of Benton et al. would attach effects to base types. It is not clear what the translation that corresponds to the call-by-name instantiation of Effect would be.

Levy [14, 15] describes call-by-push-value, and translations into it from call-by-value and call-by-name. Unlike the previous two, it does not use a monad directly, but instead uses its decomposition into an adjunction between free and forgetful functors. For call-by-name, types are interpreted as algebras of the monad. The only effect system we know of for call-by-push-value is Kammar and Plotkin's multiadjunctive intermediate language (MAIL) [16], which assumes that the effect algebra $\mathcal{F}$ is a semilattice. It would be interesting to determine if there is a translation from Name (and Value) into MAIL for these cases. It is not clear if MAIL generalizes to other effect algebras. Finally, it does not

appear that call-by-need can be translated into call-by-push-value *at all*. Hence we cannot expect call-by-push-value to be a suitable setting for modelling call-by-need effects.

Type-based analysis for lazy languages Very little has been done on type-based analysis for lazy languages. A system that determines how many times each variable has been used was created by Turner and Wadler [17]. Only minor modifications to the effect system presented here would be needed to support this use case. Wansbrough and Peyton Jones [18] later extended Turner and Wadler's work to support polymorphism and datatypes. Adding datatypes to the type systems presented here would be interesting future work.

Operational semantics of lazy languages There has been a large amount of work on operational semantics for lazy languages. The heap-based operational semantics given in Section 4.1 is based on that of Launchbury [6]. Several heapless semantics have also been described [7, 19, 20, 21]. It would be interesting to consider how the discussion of soundness in Section 4.1 can be adapted to a heapless semantics. Finally, Garcia et al. [22] discuss an abstract machine for call-by-need.

Type-based strictness analysis Work on strictness analysis [23] has focused mainly on abstract interpretation. Schrijvers and Mycroft [24] however describe a type-based strictness analysis. This is similar to the type system described here: it determines the possible traces of programs. However, they do not relate the type system to general effect systems, and only cover a first-order language. We believe that VARIABLE can be used to provide a type-based strictness analysis for higher-order call-by-need programs.

Denotational semantics of lazy languages As far as we know, there are no denotational semantics that model call-by-need languages with effects. Launchbury [6] does relate his operational semantics to a denotational semantics by proving an adequacy theorem, but the language has no side-effects. The denotational semantics more closely models call-by-name, so we would expect adequacy to fail if effects are added. Maraist et al. [25] show how to translate call-by-need into an affine calculus, which suggests it may be possible to model call-by-need using a comonad [26]. It appears that more structure than just a monad would be necessary to model the fact that the behaviour of variables changes as a program executes in a call-by-need language. Joinads [27] may be the answer to this.

8 Conclusions

In this work, we have filled a gap in the previous work on effect systems by showing that effect systems do not need to be limited to call-by-value languages. Effect systems work for languages that use other reduction strategies. In particular, we have given an effect system for call-by-need. This allows the previous work that has been done on effect systems to be applied to call-by-need languages, instead of just call-by-value languages. Our effect system should aid in reasoning about effects in lazy languages.

The main difficulty in designing an effect system is determining when the arguments are evaluated. This is non-trivial, partly because it depends on the order in which variables are used. The coeffect system Variable solves this problem without considering the effects of programs.

To give an effect system that works for multiple strategies, we modified Variable to give Effect, which includes effects in coeffects in traces as well as variable names. Effect is general: it only assumes that the effect algebra is a preordered monoid and works for several reduction strategies. It should therefore be possible to use it for a wide range of applications.

Acknowledgements This work was supported by an EPSRC studentship.

References

[1] Petricek T., Orchard D., Mycroft A., Coeffects: Unified Static Analysis of Context-dependence, In: Proceedings of the 40th International Conference on Automata, Languages, and Programming, Springer-Verlag, 2013, 385–397

[2] Katsumata S.y., Parametric Effect Monads and Semantics of Effect Systems, In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2014, 633–645

[3] Lucassen J.M., Gifford D.K., Polymorphic Effect Systems, In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1988, 47–57

[4] Hoffmann J., Das A., Weng S.C., Towards Automatic Resource Bound Analysis for OCaml, In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, ACM, 2017, 359–373

[5] Petricek T., Orchard D., Mycroft A., Coeffects: A Calculus of Context-

dependent Computation, In: Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming, ACM, 2014, 123–135

[6] Launchbury J., A Natural Semantics for Lazy Evaluation, In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1993, 144–154

[7] Ariola Z.M., Maraist J., Odersky M., Felleisen M., Wadler P., A Call-by-need Lambda Calculus, In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1995, 233–246

[8] Mycroft A., Orchard D., Petricek T., Effect Systems Revisited—Control-Flow Algebra and Semantics, In: Semantics, Logics, and Calculi, Springer, 2016, 1–32

[9] Nielson F., Nielson H.R., Type and Effect Systems, In: Correct System Design, Recent Insight and Advances, Springer-Verlag, 1999, 114–136

[10] Orchard D., Yoshida N., Effects As Sessions, Sessions As Effects, In: Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2016, 568–581

[11] Gaboardi M., Katsumata S.y., Orchard D., Breuvart F., Uustalu T., Combining Effects and Coeffects via Grading, In: Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ACM, 2016, 476–489

[12] Moggi E., Notions of Computation and Monads, Inf. Comput., 93(1), 1991, 55–92

[13] Benton N., Hughes J., Moggi E., Monads and Effects, In: Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures, Springer-Verlag, 2002, 42–122

[14] Levy P.B., Call-by-Push-Value: A Subsuming Paradigm, In: Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications, Springer-Verlag, 1999, 228–242

[15] Levy P.B., Call-by-push-value: Decomposing Call-by-value and Call-by-name, Higher Order Symbol. Comput., 19(4), 2006, 377–414

[16] Kammar O., Plotkin G.D., Algebraic Foundations for Effect-dependent Optimisations, In: Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2012, 349–360

[17] Turner D.N., Wadler P., Mossin C., Once Upon a Type, In: Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture, ACM, 1995, 1–11

[18] Wansbrough K., Peyton Jones S., Once Upon a Polymorphic Type, In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1999, 15–28

[19] Ariola Z.M., Felleisen M., The call-by-need lambda calculus, J. Functional Programming, 7(3), 1997, 265–301

[20] Chang S., Felleisen M., The Call-by-need Lambda Calculus, Revisited, In: Proceedings of the 21st European Conference on Programming Languages and Systems, Springer-Verlag, 2012, 128–147

[21] Maraist J., Odersky M., Wadler P., The Call-by-need Lambda Calculus, J. Functional Programming, 8(3), 1998, 275–317

[22] Garcia R., Lumsdaine A., Sabry A., Lazy Evaluation and Delimited Control, In: Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 2009, 153–164

[23] Mycroft A., The Theory and Practice of Transforming Call-by-need into Call-by-value, In: Proceedings of the Fourth International Symposium on Programming, Springer-Verlag, 1980, 269–281

[24] Schrijvers T., Mycroft A., Strictness Meets Data Flow, In: Proceedings of the 17th International Conference on Static Analysis, Springer-Verlag, 2010, 439–454

[25] Maraist J., Odersky M., Turner D.N., Wadler P., Call-by-name, Call-by-value, Call-by-need, and the Linear Lambda Calculus, In: Proceedings of the Eleventh Annual Mathematical Foundations of Programming Semantics Conference, 1995, 370–392

[26] Seely R., Linear Logic, *-Autonomous Categories and Cofree Coalgebras, In: Categories in Computer Science and Logic, American Mathematical Society, 1989, 371–382

[27] Petricek T., Syme D., Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming, In: R. Rocha, J. Launchbury, eds., Proceedings of Practical Aspects of Declarative Languages, Springer, Berlin, Heidelberg, 2011, 205–219