Extended call-by-push-value: reasoning about effectful programs and evaluation order

Dylan McDermott and Alan Mycroft

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

Abstract. Traditionally, reasoning about programs under varying evaluation regimes (call-by-value, call-by-name etc.) was done at the meta-level, treating them as term rewriting systems. Levy's call-by-push-value (CBPV) calculus provides a more powerful approach for reasoning, by treating CBPV terms as a common intermediate language which captures both call-by-value and call-by-name, and by allowing equational reasoning about changes to evaluation order between or within programs. We extend CBPV to additionally deal with call-by-need, which is non-trivial because of shared reductions. This allows the equational reasoning to also support call-by-need. As an example, we then prove that call-by-need and call-by-name are equivalent if nontermination is the only side-effect in the source language.

We then show how to incorporate an effect system. This enables us to exploit static knowledge of the potential effects of a given expression to augment equational reasoning; thus a program fragment might be invariant under change of evaluation regime only because of knowledge of its effects.

Keywords: Evaluation order · Call-by-need · Call-by-push-value · Logical relations · Effect systems

1 Introduction

Programming languages based on the λ-calculus have different semantics depending on the reduction strategy employed. Three common variants are call-by-value, call-by-name and call-by-need (with the third sometimes also referred to as "lazy evaluation" when data constructors defer evaluation of arguments until the data structure is traversed). Reasoning about such programs and their equivalence under varying reduction strategies can be difficult as we have to reason about meta-level reduction strategies and not merely at the object level.

Levy [17] introduced *call-by-push-value* (CBPV) to improve the situation. CBPV is a calculus with separated notions of value and computation. A characteristic feature is that each CBPV program encodes its own evaluation order. It is best seen as an *intermediate language* into which lambda-calculus-based *source-language* programs can be translated. Moreover, CBPV is powerful enough that programs employing call-by-value or call-by-name (or even a mixture) can be simply translated into it, giving an object-calculus way to reason about the meta-level concept of reduction order.

However, CBPV does not enable us to reason about call-by-need evaluation. An intuitive reason is that call-by-need has "action at a distance" in that reduction of one subterm causes reduction of all other subterms that originated as copies during variable substitution. Indeed call-by-need is often framed using mutable stores (graph reduction [32], or reducing a thunk which is accessed by multiple pointers [16]). CBPV does not allow these to be encoded.

This work presents *extended call-by-push-value* (ECBPV), a calculus similar to CBPV, but which can capture call-by-need reduction in addition to call-by-value and call-by-name. Specifically, ECBPV adds an extra primitive $M\,\mathsf{need}\,\underline{x}.\,N$ which runs N, with M being evaluated the first time $\underline{x}$ is used. On subsequent uses of $\underline{x}$, the result of the first run is returned immediately. The term M is evaluated at most once. We give the syntax and type system of ECBPV, together with an equational theory that expresses when terms are considered equal.

A key justification for an intermediate language that can express several evaluation orders is that it enables equivalences between the evaluation orders to be proved. If there are no (side-)effects at all in the source language, then call-by-need, call-by-value and call-by-name should be semantically equivalent. If the only effect is nondeterminism, then need and value (but not name) are equivalent. If the only effect is nontermination then need and name (but not value) are equivalent. We show that ECBPV can be used to prove such equivalences by proving the latter using an argument based on *Kripke logical relations of varying arity* [12].

These equivalences rely on the *language* being restricted to particular effects. However, one may wish to switch evaluation order for *subprograms* restricted to particular effects, even if the language itself does not have such a restriction. To allow reasoning to be applied to these cases, we add an *effect system* [20] to ECBPV, which allows the side-effects of subprograms to be statically estimated. This allows us to determine which parts of a program are invariant under changes in evaluation order. As we will see, support for call-by-need (and action at a distance more generally) makes describing an effect system significantly more difficult than for call-by-value.

Contributions We make the following contributions:

- We describe *extended call-by-push-value*, a version of CBPV containing an extra construct that adds support for call-by-need. We give its syntax, type system, and equational theory (Section 2).
- We describe two translations from a lambda-calculus source language into ECBPV: one for call-by-name and one for call-by-need (the first such translation) (Section 3). We then show that, if the source language has nontermination as the only effect, call-by-name and call-by-need are equivalent.
- We refine the type system of ECBPV so that its types also carry effect information (Section 4). This allows equivalences between evaluation orders to be exploited, both at ECBPV and source level, when subprograms are statically limited to particular effects.

2 Extended call-by-push-value

We describe an extension to call-by-push-value with support for call-by-need. The primary difference between ordinary CBPV and ECBPV is the addition of a primitive that allows computations to be added to the environment, so that they are evaluated only the first time they are used. Before describing this change, we take a closer look at CBPV and how it supports call-by-value and call-by-name.

CBPV stratifies terms into *values*, which do not have side-effects, and *computations*, which might. Evaluation order is irrelevant for values, so we are only concerned with how computations are sequenced. There is exactly one primitive that causes the evaluation of more than one computation, which is the computation $M\,\mathsf{to}\,x.\,N$. This means run the computation M, bind the result to x, and then run the computation N. (It is similar to `M >>= \x -> N` in Haskell.) The evaluation order is fixed: M is always eagerly evaluated. This construct can be used to implement call-by-value: to apply a function, eagerly evaluate the argument and then evaluate the body of the function. No other constructs cause the evaluation of more than one computation.

To allow more control over evaluation order, CBPV allows computations to be thunked. The term $\mathsf{thunk}\,M$ is a value that contains the thunk of the computation M. Thunks can be duplicated (to allow a single computation to be evaluated more than once), and can be converted back into computations with $\mathsf{force}\,V$. This allows call-by-name to be implemented: arguments to functions are thunked computations. Arguments are used by forcing them, so that the computation is evaluated every time the argument is used. Effectively, there is a construct $M\,\mathsf{name}\,\underline{x}.\,N$, which evaluates M each time the variable $\underline{x}$ is used by N, rather than eagerly evaluating. (The variable $\underline{x}$ is underlined here to indicate that it refers to a computation rather than a value: uses of it may have side-effects.)

To support call-by-need, extended call-by-push-value adds another construct $M\,\mathsf{need}\,\underline{x}.\,N$. This term runs the computation N, with the computation M being evaluated the first time $\underline{x}$ is used. On subsequent uses of $\underline{x}$, the result of the first run is returned immediately. The computation M is evaluated at most once. This new construct adds the “action at a distance” missing from ordinary CBPV.

We briefly mention that adding general mutable references to call-by-push-value would allow call-by-need to be encoded. However, reasoning about evaluation order would be difficult, and so we do not take this option.

2.1 Syntax

The syntax of extended call-by-push-value is given in Figure 1. The highlighted parts are new here. The rest of the syntax is similar to CBPV.[1]

[1] The only difference is that eliminators of product and sum types are value terms rather than computation terms (which makes value terms slightly more general). Levy [17] calls this CBPV with *complex values*.

$$\begin{aligned}
V, W &::= c \mid x \mid (V_1, V_2) \mid \mathsf{fst}\, V \mid \mathsf{snd}\, V \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V \\
&\quad\mid \mathsf{case}\, V\, \mathsf{of}\, \{\mathsf{inl}\, x.\, W_1, \mathsf{inr}\, y.\, W_2\} \mid \mathsf{thunk}\, M \\
M, N &::= \underline{x} \mid \mathsf{force}\, V \mid \lambda\{i.\, M_i\}_{i \in I} \mid i`M \mid \lambda x.\, M \mid V`M \mid \mathsf{return}\, V \\
&\quad\mid M\, \mathsf{to}\, x.\, N \mid M\, \mathsf{need}\, \underline{x}.\, N \\
A, B &::= \mathbf{unit} \mid A_1 \times A_2 \mid A_1 + A_2 \mid \mathbf{U}\, \underline{C} \\
\underline{C}, \underline{D} &::= \textstyle\prod_{i \in I} \underline{C}_i \mid A \to \underline{C} \mid \mathbf{Fr}\, A \\
\Gamma &::= \diamond \mid \Gamma, x : A \mid \Gamma, \underline{x} : \mathbf{Fr}\, A
\end{aligned}$$

Fig. 1: Syntax of ECBPV

We assume two sets of variables: *value variables* $x, y, \ldots$ and *computation variables* $\underline{x}, \underline{y}, \ldots$. While ordinary CBPV does not include computation variables, they do not of themselves add any expressive power to the calculus. The ability to use call-by-need in ECBPV comes from the need construct used to bind the variable.[2]

There are two kinds of terms, *value terms* V, W which do not have side-effects (in particular, are strongly normalizing), and *computation terms* M, N which might have side-effects. Value terms include constants c, and specifically the constant $()$ of type **unit**. There are no constant computation terms; value constants suffice (see Section 3 for an example). The value term $\mathsf{thunk}\, M$ suspends the computation M; the computation term $\mathsf{force}\, V$ runs the suspended computation V. Computation terms also include I-ary tuples $\lambda\{i.\, M_i\}_{i \in I}$ (where I ranges over *finite* sets); the ith projection of a tuple M is $i`M$. Functions send values to computations, and are computations themselves. Application is written $V`M$, where V is the argument and M is the function to apply. The term $\mathsf{return}\, V$ is a computation that just returns the value V, without causing any side-effects. Eager sequencing of computations is given by $M\, \mathsf{to}\, x.\, N$, which evaluates M until it returns a value, then places the result in x and evaluates N. For example, in $M\, \mathsf{to}\, x.\ \mathsf{return}\, (x, x)$, the term M is evaluated once, and the result is duplicated. In $M\, \mathsf{to}\, x.\ \mathsf{return}\, ()$, the term M is still evaluated once, but its result is never used. Syntactically, both to and need (explained below) are right-associative (so $M_1\, \mathsf{to}\, x.\, M_2\, \mathsf{to}\, y.\, M_3$ means $M_1\, \mathsf{to}\, x.\, (M_2\, \mathsf{to}\, y.\, M_3)$).

The primary new construct is $M\, \mathsf{need}\, \underline{x}.\, N$. This term evaluates N. The first time $\underline{x}$ is evaluated (due to a use of $\underline{x}$ inside N) it behaves the same as the computation M. If M returns a value V, then subsequent uses of $\underline{x}$ behave the same as $\mathsf{return}\, V$. Hence only the first use of $\underline{x}$ will evaluate M. If $\underline{x}$ is not used then M is not evaluated at all. The computation variable $\underline{x}$ bound inside the term is primarily used by eagerly sequencing it with other computations. For

[2] Computation variables are not strictly required to support call-by-need (since we can use $x : \mathbf{U}\, (\mathbf{Fr}\, A)$ instead of $\underline{x} : \mathbf{Fr}\, A$), but they simplify reasoning about evaluation order, and therefore we choose to include them.

example,

$$M\ \mathsf{need}\ \underline{x}.\ \underline{x}\ \mathsf{to}\ y.\ \underline{x}\ \mathsf{to}\ z.\ \mathsf{return}\,(y, z)$$

uses $\underline{x}$ twice: once where the result is bound to y, and once where the result is bound to z. Only the first of these uses will evaluate M, so this term has the same semantics as M to x. $\mathsf{return}(x, x)$. The term M need $\underline{x}$. $\mathsf{return}\,()$ does not evaluate M at all, and has the same semantics as $\mathsf{return}\,()$.

With the addition of need it is not in general possible to determine the order in which computations are executed statically. Uses of computation variables are given statically, but not all of these actually evaluate the corresponding computation dynamically. In general, the set of uses of computation variables that actually cause effects depends on run-time behaviour. This will be important when describing the effect system in Section 4.

The standard capture-avoiding substitution of value variables in value terms is denoted $V[x \mapsto W]$. We similarly have substitutions of value variables in computation terms, computation variables in value terms, and computation variables in computation terms. Finally, we define the call-by-name construct mentioned above as syntactic sugar for other CBPV primitives:

$$M\ \mathsf{name}\ \underline{x}.\, N \ \coloneqq\ \mathsf{thunk}\, M\ \text{`}\ \lambda y.\, N[\underline{x} \mapsto \mathsf{force}\, y]$$

where y is not free in N.

Types are stratified into *value types* A, B and *computation types* $\underline{C}, \underline{D}$. Value types include the unit type, products and sum types. (It is easy to add further base types; we omit Levy's empty types for simplicity.) Value types also include *thunk types* $\mathbf{U}\,\underline{C}$, which are introduced by $\mathsf{thunk}\, M$ and eliminated by $\mathsf{force}\, V$. Computation types include I-ary product types $\prod_{i \in I} \underline{C}_i$ for finite I, function types $A \to \underline{C}$, and *returner types* $\mathbf{Fr}\, A$. The latter are introduced by $\mathsf{return}\, V$, and are the only types of computation that can appear on the left of either to or need (which are the eliminators of returner types). The type constructors $\mathbf{U}$ and $\mathbf{Fr}$ form an *adjunction* in categorical models. Finally, contexts Γ map value variables to value types, and computation variables to computation types of the form $\mathbf{Fr}\, A$. This restriction is due to the fact that the only construct that binds computation variables is need, which only sequences computations of returner type. Allowing computation variables to be associated with other forms of computation type in typing contexts is therefore unnecessary. Typing contexts are ordered lists.

The syntax is parameterized by a *signature*, containing the constants c.

Definition 1 (Signature). *A* signature $\mathcal{K}$ *consists of a set* $\mathcal{K}_A$ *of constants of type* A *for each value type* A*. All signatures contain* $() \in \mathcal{K}_{\mathbf{unit}}$.

2.2 Type system

The type system of extended call-by-push-value is a minor extension of the type system of ordinary call-by-push-value. Assume a fixed signature $\mathcal{K}$. There are two typing judgements, one for value types and one for computation types. The

$$\boxed{\Gamma \vdash_{\mathrm{v}} V : A}$$

$$\frac{}{\Gamma \vdash_{\mathrm{v}} x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash_{\mathrm{v}} c : A}\ \text{if } c \in \mathcal{K}_A \qquad \frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash_{\mathrm{v}} \mathsf{thunk}\, M : \mathbf{U}\, \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V_1 : A_1 \qquad \Gamma \vdash_{\mathrm{v}} V_2 : A_2}{\Gamma \vdash_{\mathrm{v}} (V_1, V_2) : A_1 \times A_2} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A_1 \times A_2}{\Gamma \vdash_{\mathrm{v}} \mathsf{fst}\, V : A_1} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A_1 \times A_2}{\Gamma \vdash_{\mathrm{v}} \mathsf{snd}\, V : A_2}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A_1}{\Gamma \vdash_{\mathrm{v}} \mathsf{inl}\, V : A_1 + A_2} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A_2}{\Gamma \vdash_{\mathrm{v}} \mathsf{inr}\, V : A_1 + A_2}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A_1 + A_2 \qquad \Gamma, x : A_1 \vdash_{\mathrm{v}} W_1 : B \qquad \Gamma, x : A_2 \vdash_{\mathrm{v}} W_2 : B}{\Gamma \vdash_{\mathrm{v}} \mathsf{case}\, V\ \mathsf{of}\ \{\mathsf{inl}\, x.\, W_1, \mathsf{inr}\, y.\, W_2\} : B}$$

$$\boxed{\Gamma \vdash M : \underline{C}}$$

$$\frac{}{\Gamma \vdash \underline{x} : \mathbf{Fr}\, A}\ \text{if } (\underline{x} : \mathbf{Fr}\, A) \in \Gamma \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Gamma \vdash \mathsf{return}\, V : \mathbf{Fr}\, A} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : \mathbf{U}\, \underline{C}}{\Gamma \vdash \mathsf{force}\, V : \underline{C}}$$

$$\frac{(\Gamma \vdash M_i : \underline{C}_i)_{i \in I}}{\Gamma \vdash \lambda\{i.\, M_i\}_{i \in I} : \prod_{i \in I} \underline{C}_i} \qquad \frac{\Gamma \vdash M : \prod_{i \in I} \underline{C}_i}{\Gamma \vdash i\text{‘}M : \underline{C}_i}$$

$$\frac{\Gamma, x : A \vdash M : \underline{C}}{\Gamma \vdash \lambda x.\, M : A \to \underline{C}} \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A \qquad \Gamma \vdash M : A \to \underline{C}}{\Gamma \vdash V\text{‘}M : \underline{C}}$$

$$\frac{\Gamma \vdash M : \mathbf{Fr}\, A \qquad \Gamma, x : A \vdash N : \underline{C}}{\Gamma \vdash M\ \mathsf{to}\ x.\, N : \underline{C}} \qquad \frac{\Gamma \vdash M : \mathbf{Fr}\, A \qquad \Gamma, \underline{x} : \mathbf{Fr}\, A \vdash N : \underline{C}}{\Gamma \vdash M\ \mathsf{need}\ \underline{x}.\, N : \underline{C}}$$

Fig. 2: Typing rules for ECBPV

rules for the value typing judgement $\Gamma \vdash_{\mathrm{v}} V : A$ and the computation typing judgement $\Gamma \vdash M : \underline{C}$ are given in Figure 2. Rules that add a new variable to the typing context implicitly require that the variable does not already appear in the context. The type system admits the usual weakening and substitution properties for both value and computation variables.

It should be clear that ECBPV is actually an extension of call-by-push-value. CBPV terms embed as terms that never use the highlighted forms. We translate call-by-need by encoding call-by-need functions as terms of the form

$$\lambda x'.\, (\mathsf{force}\, x')\ \mathsf{need}\ \underline{x}.\, M$$

where x' is not free in M. This is a call-by-push-value function that accepts a thunk as an argument. The thunk is added to the context, and the body of the function is executed. The first time the argument is used (via $\underline{x}$), the computation inside the thunk is evaluated. Subsequent uses do not run the computation again.

A translation based on this idea from a call-by-need source language is given in detail in Section 3.2.

2.3 Equational theory

In this section, we present the *equational theory* of extended call-by-push-value. This is an extension of the equational theory for CBPV given by Levy [17] to support our new constructs. It consists of two judgement forms, one for values and one for computations:

$$\Gamma \vdash_{\mathrm{v}} V \equiv W : A \qquad \Gamma \vdash M \equiv N : \underline{C}$$

These mean both terms are well typed, and are considered equal by the equational theory. We frequently omit the context and type when they are obvious or unimportant.

The definition is given by the axioms in Figure 3. Note that these axioms only hold when the terms they mention have suitable types, and when suitable constraints on free variables are satisfied. For example, the second sequencing axiom holds only if $\underline{x}$ is not free in N. These conditions are left implicit in the figure. The judgements are additionally reflexive (assuming the typing holds), symmetric and transitive. They are also closed under all possible congruence rules. There are no restrictions on congruence related to evaluation order. None are necessary because ECBPV terms make the evaluation order explicit: all sequencing of computations uses to and need. Finally, note that enriching the signature with additional constants will in general require additional axioms capturing their behaviour; Section 3 exemplifies this for constants $\bot_A$ representing nontermination.

For the equational theory to capture call-by-need, we might expect computation terms that are not of the form $\mathsf{return}\, V$ to never be duplicated, since they should not be evaluated more than once. There are two exceptions to this. Such terms can be duplicated in the axioms that duplicate value terms (such as the β laws for sum types). In this case, the syntax ensures such terms are thunked. This is correct because we should allow these terms to be executed once in each separate execution of a computation (and separate executions arise from duplication of thunks). We are only concerned with duplication *within* a single computation. Computation terms can also be duplicated across multiple elements of a tuple $\lambda\{i.\, M_i\}$ of computation terms. This is also correct, because only one component of a tuple can be used within a single computation (without thunking), so the effects still will not happen twice. (There is a similar consideration for functions, which can only be applied once.) The remainder of the axioms never duplicate need-bound terms that might have effects.

The majority of the axioms of the equational theory are standard. Only the axioms involving need are new; these are highlighted. The first new sequencing axiom (in Figure 3c) is the crucial one. It states that if a computation will next evaluate $\underline{x}$, where $\underline{x}$ is a computation variable bound to M, then this is the same as evaluating M, and then using the result for subsequent uses of $\underline{x}$. In particular, this axiom (together with the η law for $\mathbf{Fr}$) implies that $M \;\mathsf{need}\; \underline{x}.\, \underline{x} \;\equiv\; M$.

$$
\begin{aligned}
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{fst}\,(V_1, V_2) &\equiv V_1 \ : \ A_1 \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{snd}\,(V_1, V_2) &\equiv V_2 \ : \ A_2 \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{case}\,\mathsf{inl}\,V\ \mathsf{of}\ \{\mathsf{inl}\,x.\,W_1, \mathsf{inr}\,y.\,W_2\} &\equiv W_1[x \mapsto V] \ : \ B \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{case}\,\mathsf{inr}\,V\ \mathsf{of}\ \{\mathsf{inl}\,x.\,W_1, \mathsf{inr}\,y.\,W_2\} &\equiv W_2[y \mapsto V] \ : \ B \\
\mathit{\Gamma} \vdash \ \mathsf{force}(\mathsf{thunk}\,M) &\equiv M \ : \ \underline{C} \\
\mathit{\Gamma} \vdash \ i\,\text{‘}\,\lambda\{i.\,M_i\}_{i \in I} &\equiv M_i \ : \ \underline{C}_i \\
\mathit{\Gamma} \vdash \ V\,\text{‘}\,\lambda x.\,M &\equiv M[x \mapsto V] \ : \ \underline{C} \\
\mathit{\Gamma} \vdash \ \mathsf{return}\,V\ \mathsf{to}\ x.\,M &\equiv M[x \mapsto V] \ : \ \underline{C} \\
\mathit{\Gamma} \ \vdash \ \mathsf{return}\,V\ \mathsf{need}\ \underline{x}.\,M &\equiv M[\underline{x} \mapsto \mathsf{return}\,V] \ : \ \underline{C}
\end{aligned}
$$

(a) β laws

$$
\begin{aligned}
\mathit{\Gamma} \vdash_{\mathrm{v}} \ () &\equiv V \ : \ \mathbf{unit} \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ (\mathsf{fst}\,V, \mathsf{snd}\,V) &\equiv V \ : \ A_1 \times A_2 \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{case}\,W\ \mathsf{of}\ \{\mathsf{inl}\,y.\,V[x \mapsto \mathsf{inl}\,y], \mathsf{inr}\,z.\,V[x \mapsto \mathsf{inr}\,z]\} &\equiv V[x \mapsto W] \ : \ B \\
\mathit{\Gamma} \vdash_{\mathrm{v}} \ \mathsf{thunk}(\mathsf{force}\,M) &\equiv M \ : \ \mathbf{U}\,\underline{C} \\
\mathit{\Gamma} \vdash \ \lambda\{i.\,i\,\text{‘}\,M\}_{i \in I} &\equiv M \ : \ \textstyle\prod_{i \in I} \underline{C}_i \\
\mathit{\Gamma} \vdash \ \lambda x.\,x\,\text{‘}\,M &\equiv M \ : \ A \to \underline{C} \\
\mathit{\Gamma} \vdash \ M\ \mathsf{to}\ x.\ \mathsf{return}\,x &\equiv M \ : \ \mathbf{Fr}\,A
\end{aligned}
$$

(b) η laws

$$
\begin{aligned}
\mathit{\Gamma} \ \vdash \ M\ \mathsf{need}\ \underline{x}.\,\underline{x}\ \mathsf{to}\ y.\,N &\equiv M\ \mathsf{to}\ y.\,N[\underline{x} \mapsto \mathsf{return}\,y] \ : \ \underline{C} \\
\mathit{\Gamma} \ \vdash \ M\ \mathsf{need}\ \underline{x}.\,N &\equiv N \ : \ \underline{C}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{\Gamma} \vdash \ \lambda\{i.\,M\ \mathsf{to}\ x.\,N_i\}_{i \in I} &\equiv M\ \mathsf{to}\ x.\,\lambda\{i.\,N_i\}_{i \in I} \ : \ \textstyle\prod_{i \in I} \underline{C}_i \\
\mathit{\Gamma} \vdash \ \lambda y.\,M\ \mathsf{to}\ x.\,N &\equiv M\ \mathsf{to}\ x.\,\lambda y.\,N \ : \ A \to \underline{C} \\
\mathit{\Gamma} \ \vdash \ \lambda\{i.\,M\ \mathsf{need}\ \underline{x}.\,N_i\}_{i \in I} &\equiv M\ \mathsf{need}\ \underline{x}.\,\lambda\{i.\,N_i\}_{i \in I} \ : \ \textstyle\prod_{i \in I} \underline{C}_i \\
\mathit{\Gamma} \ \vdash \ \lambda y.\,M\ \mathsf{need}\ \underline{x}.\,N &\equiv M\ \mathsf{need}\ \underline{x}.\,\lambda y.\,N \ : \ A \to \underline{C}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{\Gamma} \vdash \ (M_1\ \mathsf{to}\ x.\,M_2)\ \mathsf{to}\ y.\,M_3 &\equiv M_1\ \mathsf{to}\ x.\,M_2\ \mathsf{to}\ y.\,M_3 \ : \ \underline{C} \\
\mathit{\Gamma} \ \vdash \ M_1\ \mathsf{to}\ x.\,M_2\ \mathsf{need}\ \underline{y}.\,M_3 &\equiv M_2\ \mathsf{need}\ \underline{y}.\,M_1\ \mathsf{to}\ x.\,M_3 \ : \ \underline{C} \\
\mathit{\Gamma} \ \vdash \ (M_1\ \mathsf{need}\ \underline{x}.\,M_2)\ \mathsf{to}\ y.\,M_3 &\equiv M_1\ \mathsf{need}\ \underline{x}.\,M_2\ \mathsf{to}\ y.\,M_3 \ : \ \underline{C} \\
\mathit{\Gamma} \ \vdash \ (M_1\ \mathsf{need}\ \underline{x}.\,M_2)\ \mathsf{need}\ \underline{y}.\,M_3 &\equiv M_1\ \mathsf{need}\ \underline{x}.\,M_2\ \mathsf{need}\ \underline{y}.\,M_3 \ : \ \underline{C}
\end{aligned}
$$

(c) Sequencing axioms

Fig. 3: Equational theory of ECBPV

The second sequencing axiom does *garbage collection* [22]: if a computation bound by need is not used (because the variable does not appear), then the binding can be dropped. This equation implies, for example, that

$$M_1 \;\mathsf{need}\; \underline{x}_1.\, M_2 \;\mathsf{need}\; \underline{x}_2.\, \cdots M_n \;\mathsf{need}\; \underline{x}_n.\; \mathsf{return}\,() \;\;\equiv\;\; \mathsf{return}\,()$$

The next four sequencing axioms (two from CBPV and two new) state that binding a computation with to or need commutes with the remaining forms of computation terms. These allow to and need to be moved to the outside of other constructs *except* thunks. The final four axioms (one from CBPV and three new) capture associativity and commutativity involving need and to; again these parallel the existing simple associativity axiom for to.

Note that associativity between different evaluation orders is not necessarily valid. In particular, we do not have

$$(M_1 \;\mathsf{to}\; x.\, M_2) \;\mathsf{need}\; \underline{y}.\, M_3 \quad\equiv\quad M_1 \;\mathsf{to}\; x.\, (M_2 \;\mathsf{need}\; \underline{x}.\, M_3)$$

(The first term might not evaluate M_1, the second always does.) This is usually the case when evaluation orders are mixed [26].

These final two groups allow computation terms to be placed in normal forms where bindings of computations are on the outside. (Compare this with the translation of source-language *answers* given in Section 3.2.) Finally, the β law for need (in Figure 3a) parallels the usual β law for to: it gives the behaviour of computation terms that return values without having any effects.

The above equational theory induces a notion of *contextual equivalence* $\cong_{\mathrm{ctx}}$ between ECBPV terms. Two terms are contextually equivalent when they have no observable differences in behaviour. When we discuss *equivalences between evaluation orders* in Section 3, $\cong_{\mathrm{ctx}}$ is the notion of *equivalence between terms* that we consider.

Contextual equivalence is defined as follows. The *ground types* G are the value types that do not contain thunks:

$$G ::= \mathbf{unit} \mid G_1 \times G_2 \mid G_1 + G_2$$

A *value-term context* $\mathcal{C}[-]$ is a computation term with a single hole (written $-$), which occurs in a position where a value term is expected. We write $\mathcal{C}[V]$ for the computation term that results from replacing the hole with V. Similarly, *computation-term contexts* $\underline{\mathcal{C}}[-]$ are computation terms with a single hole where a computation term is expected, and $\underline{\mathcal{C}}[M]$ is the term in which the hole is replaced by M. Contextual equivalence says that the terms cannot be distinguished by closed computations that return ground types. (Recall that $\diamond$ is the empty typing context.)

Definition 2 (Contextual equivalence). *There are two judgement forms of* contextual equivalence*.*

1. *Between value terms:* $\Gamma \vdash_{\mathrm{v}} V \cong_{\mathrm{ctx}} W : A$ *if* $\Gamma \vdash_{\mathrm{v}} V : A$*,* $\Gamma \vdash_{\mathrm{v}} W : A$*, and for all ground types* G *and value-term contexts* $\mathcal{C}$ *such that* $\diamond \vdash \mathcal{C}[V] : \mathbf{Fr}\, G$ *and* $\diamond \vdash \mathcal{C}[W] : \mathbf{Fr}\, G$ *we have*

$$\diamond \vdash \mathcal{C}[V] \equiv \mathcal{C}[W] : \mathbf{Fr}\, G$$

2. *Between computation terms:* $\Gamma \vdash M \cong_{\mathrm{ctx}} N : \underline{C}$ *if* $\Gamma \vdash M : \underline{C}$, $\Gamma \vdash N : \underline{C}$, *and for all ground types* G *and computation-term contexts* $\underline{\mathcal{C}}[-]$ *such that* $\diamond \vdash \underline{\mathcal{C}}[M] : \mathbf{Fr}\,G$ *and* $\diamond \vdash \underline{\mathcal{C}}[N] : \mathbf{Fr}\,G$ *we have*

$$\diamond \vdash \underline{\mathcal{C}}[M] \equiv \underline{\mathcal{C}}[N] : \mathbf{Fr}\,G$$

3 Call-by-name and call-by-need

Extended call-by-push-value can be used to prove equivalences between evaluation orders. In this section we prove a classic example: if the only effect in the source language is nontermination, then call-by-name is equivalent to call-by-need. We do this in two stages.

First, we show that call-by-name is equivalent to call-by-need *within* ECBPV (Section 3.1). Specifically, we show that

$$M\ \mathsf{name}\ \underline{x}.\, N \ \ \cong_{\mathrm{ctx}}\ \ M\ \mathsf{need}\ \underline{x}.\, N$$

(Recall that $M\ \mathsf{name}\ \underline{x}.\, N$ is syntactic sugar for $\mathsf{thunk}\,M \mathbin{\text{‘}} \lambda y.\, N[\underline{x} \mapsto \mathsf{force}\,y]$.)

Second, an important corollary is that the meta-level reduction strategies are equivalent (Section 3.2). We show this by describing a lambda-calculus-based source language together with a call-by-name and a call-by-need operational semantics and giving sound (see Theorem 2) call-by-name and call-by-need translations into ECBPV. The former is based on the translation into the monadic metalanguage given by Moggi [25] (we expect Levy's translation [17] to work equally well). The call-by-need translation is new here, and its existence shows that ECBPV does indeed subsume call-by-need. We then show that given any source-language expression, the two translations give contextually equivalent ECBPV terms.

To model non-termination being our sole source-language effect, we use the ECBPV signature which contains a constant $\bot_A : \mathbf{U}\,(\mathbf{Fr}\,A)$ for each value type A, representing a thunked diverging computation. It is likely that our proofs still work if we have general fixed-point operators as constants, but for simplicity we do not consider this here. The constants $\bot_A$ enable us to define a diverging computation $\Omega_{\underline{C}}$ for each computation type $\underline{C}$:

$$\Omega_{\mathbf{Fr}A} \coloneqq \mathsf{force}\,\bot_A \qquad \Omega_{\prod_{i\in I}\underline{C}_i} \coloneqq \lambda\{i.\,\Omega_{\underline{C}_i}\}_{i\in I} \qquad \Omega_{A\to\underline{C}} \coloneqq \lambda x.\,\Omega_{\underline{C}}$$

We characterise nontermination by augmenting the equational theory of Section 2.3 with the axiom

$$\Gamma\ \vdash\ \Omega_{\mathbf{Fr}A}\ \mathsf{to}\ x.\, M\ \equiv\ \Omega_{\underline{C}}\ :\ \underline{C} \qquad\qquad \text{(Omega)}$$

for each context Γ, value type A and computation type $\underline{C}$. In other words, diverging as part of a larger computation causes the entire computation to diverge. This is the only change to the equational theory we need to represent nontermination. In particular, we do not add additional axioms involving need.

3.1 The equivalence at the object (internal) level

In this section, we show our primary result that

$$M \ \mathsf{name}\ \underline{x}.\, N \ \ \cong_{\mathrm{ctx}}\ \ M \ \mathsf{need}\ \underline{x}.\, N$$

As is usually the case for proofs of contextual equivalence, we use *logical relations* to get a strong enough inductive hypothesis for the proof to go through. However, unlike the usual case, it does not suffice to relate *closed* terms. To see why, consider a closed term M of the form

$$\Omega_{\mathbf{Fr}A}\ \mathsf{need}\ \underline{x}.\, N_1\ \mathsf{to}\ y.\, N_2$$

If we relate only closed terms, then we do not learn anything about N_1 itself (since $\underline{x}$ may be free in it). We could attempt to proceed by considering the closed term $\Omega_{\mathbf{Fr}A}\ \mathsf{need}\ \underline{x}.\, N_1$. For example, if this returns a value V then $\underline{x}$ cannot have been evaluated and M should have the same behaviour as $\Omega_{\mathbf{Fr}A}\ \mathsf{need}\ \underline{x}.\, N_2[y \mapsto V]$. However, we get stuck when proving the last step. This is only a problem because $\Omega_{\mathbf{Fr}A}$ is a nonterminating computation: every terminating computation of returner type has the form $\mathsf{return}\,V$ (up to $\equiv$), and when these are bound using need we can eliminate the binding using the equation

$$\mathsf{return}\,V\ \mathsf{need}\ \underline{x}.\, M \equiv M[\underline{x} \mapsto \mathsf{return}\,V]$$

The solution is to relate terms that may have free computation variables (we do not need to consider free value variables). The free computation variables should be thought of as referring to nonterminating computations (because we can remove the bindings of variables that refer to terminating computations). We relate open terms using *Kripke logical relations of varying arity*, which were introduced by Jung and Tiuryn [12] to study lambda definability.

We need a number of definitions first. A context Γ' *weakens* another context Γ, written $\Gamma' \triangleright \Gamma$, whenever Γ is a sublist of Γ'. For example, $(\Gamma, \underline{x} : \mathbf{Fr}\,A) \triangleright \Gamma$. We define $\mathrm{Term}^{\Gamma}_{A}$ as the set of equivalence classes (up to the equational theory $\equiv$) of terms of value type A in context Γ, and similarly define $\underline{\mathrm{Term}}^{\Gamma}_{\underline{D}}$ for computation types:

$$\mathrm{Term}^{\Gamma}_{A} \coloneqq \{[V]_{\equiv} \mid \Gamma \vdash_{\mathrm{v}} V : A\} \qquad \underline{\mathrm{Term}}^{\Gamma}_{\underline{D}} \coloneqq \{[M]_{\equiv} \mid \Gamma \vdash M : \underline{D}\}$$

Since weakening is admissible for both typing judgements, $\Gamma' \triangleright \Gamma$ implies that $\mathrm{Term}^{\Gamma}_{A} \subseteq \mathrm{Term}^{\Gamma'}_{A}$ and $\underline{\mathrm{Term}}^{\Gamma}_{\underline{D}} \subseteq \underline{\mathrm{Term}}^{\Gamma'}_{\underline{D}}$ (note the contravariance).

A *computation context*, ranged over by Δ, is a typing context that maps variables to computation types (i.e. has the form $\underline{x}_1 : \mathbf{Fr}\,A_1, \ldots, \underline{x}_n : \mathbf{Fr}\,A_n$). Variables in computation contexts refer to nonterminating computations for the proof of contextual equivalence. A *Kripke relation* is a family of binary relations indexed by computation contexts that respects weakening of terms:

Definition 3 (Kripke relation). *A* Kripke relation R *over a value type* A *(respectively a computation type* $\underline{D}$*) is a family of relations* $R^{\Delta} \subseteq \mathrm{Term}^{\Delta}_{A} \times \mathrm{Term}^{\Delta}_{A}$ *(respectively* $R^{\Delta} \subseteq \underline{\mathrm{Term}}^{\Delta}_{\underline{D}} \times \underline{\mathrm{Term}}^{\Delta}_{\underline{D}}$*) indexed by computation contexts* Δ *such that whenever* $\Delta' \triangleright \Delta$ *we have* $R^{\Delta} \subseteq R^{\Delta'}$*.*

Note that we consider binary relations on equivalence classes of terms because we want to relate pairs of terms up to $\equiv$ (to prove contextual equivalence). The relations we define are *partial equivalence relations* (i.e. symmetric and transitive), though we do not explicitly use this fact.

We need the Kripke relations we define over computation terms to be closed under sequencing with nonterminating computations. (For the rest of this section, we omit the square brackets around equivalence classes.)

Definition 4. *A Kripke relation R over a computation type $\underline{C}$ is* closed under sequencing *if each of the following holds:*

1. *If $(\underline{x} : \mathbf{Fr}\, A) \in \Delta$ and $M, M' \in \underline{\mathrm{Term}}_{\underline{C}}^{\Delta, y:A}$ then $(\underline{x}\ \mathsf{to}\ y.\, M, \underline{x}\ \mathsf{to}\ y.\, M') \in R^{\Delta}$.*
2. *The pair $(\Omega_{\underline{C}}, \Omega_{\underline{C}})$ is in R^{Δ}.*
3. *For all $(M, M') \in R^{\Delta, \underline{y}:\mathbf{Fr} A}$ and $N \in \{\Omega_{\mathbf{Fr} A}\} \cup \{\underline{x} \mid (\underline{x} : \mathbf{Fr}\, A) \in \Delta\}$, all four of the following pairs are in R^{Δ}:*

$$(N\ \mathsf{need}\ \underline{y}.\, M,\ N\ \mathsf{need}\ \underline{y}.\, M') \qquad (M[\underline{y} \mapsto N],\ M'[\underline{y} \mapsto N])$$
$$(M[\underline{y} \mapsto N],\ N\ \mathsf{need}\ \underline{y}.\, M') \qquad (N\ \mathsf{need}\ \underline{y}.\, M,\ M'[\underline{y} \mapsto N])$$

For the first case of the definition, recall that the computation variables in Δ refer to nonterminating computations. Hence the behaviour of M and M' are irrelevant (they are never evaluated), and we do not need to assume they are related.[3] The second case implies (using axiom Omega) that

$$(\Omega_{\mathbf{Fr} A}\ \mathsf{to}\ y.\, M, \Omega_{\mathbf{Fr} A}\ \mathsf{to}\ y.\, M') \in R^{\Delta}$$

mirroring the first case. The third case is the most important. It is similar to the first (it is there to ensure that the relation is closed under the primitives used to combine computations). However, since we are showing that need is contextually equivalent to substitution, we also want these to be related. We have to consider computation variables in the definition (as possible terms N) only because of our use of Kripke logical relations. For ordinary logical relations, there would be no free variables to consider.

The key part of the proof of contextual equivalence is the definition of the Kripke logical relation, which is a family of relations indexed by value and computation types. It is defined in Figure 4 by induction on the structure of the types. In the figure, we again omit square brackets around equivalence classes.

The definition of the logical relation on ground types (**unit**, sum types and product types) is standard. Since the only way to use a thunk is to force it, the definition on thunk types just requires the two forced computations to be related.

For returner types, we want any pair of computations that return related values to be related. We also want the relation to be closed under sequencing,

[3] This is why it suffices to consider only computation contexts. If we had to relate M to M' then we would need to consider relations between terms with free value variables.

$$\boxed{R^{\Delta}_{A} \subseteq \mathrm{Term}^{\Delta}_{A} \times \mathrm{Term}^{\Delta}_{A}}$$

$$\begin{aligned}
R^{\Delta}_{\mathbf{unit}} &\coloneqq \{((),())\}\\
R^{\Delta}_{A_1 \times A_2} &\coloneqq \{(V,V') \mid (\mathsf{fst}\,V, \mathsf{fst}\,V') \in R^{\Delta}_{A_1} \wedge (\mathsf{snd}\,V, \mathsf{snd}\,V') \in R^{\Delta}_{A_2}\}\\
R^{\Delta}_{A_1 + A_2} &\coloneqq \{(\mathsf{inl}\,V, \mathsf{inl}\,V') \mid (V,V') \in R^{\Delta}_{A_1}\} \cup \{(\mathsf{inr}\,V, \mathsf{inr}\,V') \mid (V,V') \in R^{\Delta}_{A_2}\}\\
R^{\Delta}_{\mathbf{U}\underline{C}} &\coloneqq \{(V,V') \mid (\mathsf{force}\,V, \mathsf{force}\,V') \in R^{\Delta}_{\underline{C}}\}
\end{aligned}$$

$$\boxed{R^{\Delta}_{\underline{C}} \subseteq \underline{\mathrm{Term}}^{\Delta}_{\underline{C}} \times \underline{\mathrm{Term}}^{\Delta}_{\underline{C}}}$$

$$\begin{aligned}
R_{\mathbf{Fr}\,A} &\coloneqq \text{the smallest closed-under-sequencing Kripke relation such that}\\
&\qquad\qquad (V,V') \in R^{\Delta}_{A} \;\Rightarrow\; (\mathsf{return}\,V, \mathsf{return}\,V') \in R^{\Delta}_{\mathbf{Fr}\,A}\\
R^{\Delta}_{\prod_{i \in I} \underline{C}_i} &\coloneqq \{(M,M') \mid \forall i \in I.\ (i\text{‘}M, i\text{‘}M') \in R^{\Delta}_{\underline{C}_i}\}\\
R^{\Delta}_{A \to \underline{C}} &\coloneqq \{(M,M') \mid \forall \Delta', V, V'.\ \Delta' \triangleright \Delta \wedge (V,V') \in R^{\Delta'}_{A} \;\Rightarrow\; (V\text{‘}M, V'\text{‘}M') \in R^{\Delta'}_{\underline{C}}\}
\end{aligned}$$

Fig. 4: Definition of the logical relation

in order to show the fundamental lemma (below) for to and need. We therefore define $R_{\mathbf{Fr}\,A}$ as the smallest such Kripke relation. For products of computation types the definition is similar to products of value types: we require that each of the projections are related. For function types, we require as usual that related arguments are sent to related results. For this to define a Kripke relation, we have to quantify over all computation contexts Δ' that weaken Δ, because of the contravariance of the argument.

The relations we define are Kripke relations. Using the sequencing axioms of the equational theory, and the β and η laws for computation types, we can show that $R_{\underline{C}}$ is closed under sequencing for each computation type $\underline{C}$. These facts are important for the proof of the fundamental lemma.

Substitutions are given by the following grammar:

$$\sigma ::= \diamond \mid \sigma, x \mapsto V \mid \sigma, \underline{x} \mapsto M$$

We have a typing judgement $\Delta \vdash \sigma : \Gamma$ for substitutions, meaning in the context Δ the terms in σ have the types given in Γ. This is defined as follows:

$$\frac{}{\Delta \vdash \diamond : \diamond} \qquad \frac{\Delta \vdash \sigma : \Gamma \qquad \Delta \vdash_{\mathrm{v}} V : A}{\Delta \vdash (\sigma,\, x \mapsto V) : (\Gamma, x : A)} \qquad \frac{\Delta \vdash \sigma : \Gamma \qquad \Delta \vdash M : \mathbf{Fr}\,A}{\Delta \vdash (\sigma,\, \underline{x} \mapsto M) : (\Gamma, \underline{x} : \mathbf{Fr}\,A)}$$

We write $V[\sigma]$ and $M[\sigma]$ for the applications of the substitution σ to value terms V and computation terms M. These are defined by induction on the structure of the terms. The key property of the substitution typing judgement is that if $\Delta \vdash \sigma : \Gamma$, then $\Gamma \vdash_{\mathrm{v}} V : A$ implies $\Delta \vdash_{\mathrm{v}} V[\sigma] : A$ and $\Gamma \vdash M : \underline{C}$

implies $\Delta \vdash M[\sigma] : \underline{C}$. The equational theory gives us an obvious pointwise equivalence relation $\equiv$ on well-typed substitutions. We define sets $\mathrm{Subst}_{\Gamma}^{\Delta}$ of equivalence classes of substitutions, and extend the logical relation by defining $R_{\Gamma}^{\Delta} \subseteq \mathrm{Subst}_{\Gamma}^{\Delta} \times \mathrm{Subst}_{\Gamma}^{\Delta}$:

$$\begin{aligned}
\mathrm{Subst}_{\Gamma}^{\Delta} &\coloneqq \{[\sigma]_{\equiv} \mid \Delta \vdash \sigma : \Gamma\} \\
R_{\diamond}^{\Delta} &\coloneqq \{(\diamond, \diamond)\} \\
R_{\Gamma, x:A}^{\Delta} &\coloneqq \{((\sigma,\, x \mapsto V), (\sigma',\, x \mapsto V')) \mid (\sigma, \sigma') \in R_{\Gamma}^{\Delta} \wedge (V, V') \in R_{A}^{\Delta}\} \\
R_{\Gamma, \underline{x}:\mathbf{Fr} A}^{\Delta} &\coloneqq \{((\sigma,\, \underline{x} \mapsto M), (\sigma',\, \underline{x} \mapsto M')) \mid (\sigma, \sigma') \in R_{\Gamma}^{\Delta} \wedge (M, M') \in R_{\mathbf{Fr} A}^{\Delta}\}
\end{aligned}$$

As usual, the logical relations satisfy a *fundamental lemma.*

Lemma 1 (Fundamental).

1. *For all value terms $\Gamma \vdash_{\mathrm{v}} V : A$,*
$$(\sigma, \sigma') \in R_{\Gamma}^{\Delta} \quad \Rightarrow \quad (V[\sigma], V[\sigma']) \in R_{A}^{\Delta}$$
2. *For all computation terms $\Gamma \vdash M : \underline{C}$,*
$$(\sigma, \sigma') \in R_{\Gamma}^{\Delta} \quad \Rightarrow \quad (M[\sigma], M[\sigma']) \in R_{\underline{C}}^{\Delta}$$

The proof is by induction on the structure of the terms. We use the fact that each $R_{\underline{C}}$ is closed under sequencing for the to and need cases. For the latter, we also use the fact that the relations respect weakening of terms.

We also have the following two facts about the logical relation. The first roughly is that name is related to need by the logical relation, and is true because of the additional pairs that are related in the definition of closed-under-sequencing (Definition 4).

Lemma 2. *For all computation terms $\Gamma \vdash M : \mathbf{Fr}\, A$ and $\Gamma, \underline{x} : \mathbf{Fr}\, A \vdash N : \underline{C}$ we have*

$$(\sigma, \sigma') \in R_{\Gamma}^{\Delta} \quad \Rightarrow \quad ((N[\underline{x} \mapsto M])[\sigma], (M \;\mathsf{need}\; \underline{x}.\, N)[\sigma']) \in R_{\underline{C}}^{\Delta}$$

The second fact is that related terms are contextually equivalent.

Lemma 3.

1. *For all value terms $\Gamma \vdash_{\mathrm{v}} V : A$ and $\Gamma \vdash_{\mathrm{v}} V' : A$, if $(V[\sigma], V'[\sigma']) \in R_{A}^{\Delta}$ for all $(\sigma, \sigma') \in R_{\Gamma}^{\Delta}$ then*
$$\Gamma \vdash_{\mathrm{v}} V \cong_{\mathrm{ctx}} V' : A$$
2. *For all computation terms $\Gamma \vdash M : \underline{C}$ and $\Gamma \vdash M' : \underline{C}$, if $(M[\sigma], M'[\sigma']) \in R_{\underline{C}}^{\Delta}$ for all $(\sigma, \sigma') \in R_{\Gamma}^{\Delta}$ then*
$$\Gamma \vdash M \cong_{\mathrm{ctx}} M' : \underline{C}$$

This gives us enough to achieve the goal of this section.

Theorem 1. *For all computation terms $\Gamma \vdash M : \mathbf{Fr}\, A$ and $\Gamma, \underline{x} : \mathbf{Fr}\, A \vdash N : \underline{C}$, we have*

$$\Gamma \vdash M \;\mathsf{name}\; \underline{x}.\, N \cong_{\mathrm{ctx}} M \;\mathsf{need}\; \underline{x}.\, N : \underline{C}$$

3.2 The meta-level equivalence

In this section, we show that the equivalence between call-by-name and call-by-need also holds on the meta-level; this is a consequence of the object-level theorem, rather than something that is proved from scratch as it would be in a term rewriting system.

To do this, we describe a simple lambda-calculus-based source language with divergence as the only side-effect and give it a call-by-name and a call-by-need operational semantics. We then describe two translations from the source language into ECBPV. The first is a call-by-name translation based on the embedding of call-by-name in Moggi's [25] monadic metalanguage. The second is a call-by-need translation that uses our new constructs. The latter witnesses the fact that ECBPV does actually support call-by-need. Finally, we show that the two translations give contextually equivalent ECBPV terms.

The syntax, type system and operational semantics of the source language are given in Figure 5. Most of this is standard. We include only booleans and function types for simplicity. In expressions, we include a constant $\mathsf{diverge}_A$ for each type A, representing a diverging computation. (As before, it should not be difficult to replace these with general fixed-point operators.) In typing contexts, we assume that all variables are distinct, and omit the required side-condition from the figure. There is a single set of variables $x, y, \ldots$; we implicitly map these to ECBPV value or computation variables as required.

The call-by-name operational semantics is straightforward; its small-step reductions are written $e \overset{\text{name}}{\rightsquigarrow} e'$.

The call-by-need operational semantics is based on Ariola and Felleisen [2]. The only differences between the source language and Ariola and Felleisen's calculus are the addition of booleans, $\mathsf{diverge}_A$, and a type system. It is likely that we can translate other call-by-need calculi, such as those of Launchbury [16] and Maraist et al. [22]. Call-by-need small-step reductions are written $e \overset{\text{need}}{\rightsquigarrow} e'$.

The call-by-need semantics needs some auxiliary definitions. An *evaluation context* $E[-]$ is a source-language expression with a single hole, picked from the grammar given in the figure. The hole in an evaluation context indicates where reduction is currently taking place: it says which part of the expression is currently *needed*. We write $E[e]$ for the expression in which the hole is replaced with e. A (source-language) *value* is the result of a computation (the word value should not be confused with the value terms of extended call-by-push-value). An *answer* is a value in some *environment*, which maps variables to expressions. These can be thought of as *closures*. The environment is encoded in an answer using application and lambda abstraction: the answer $(\lambda x.\, a)\, e$ means the answer a where the environment maps x to e. Encoding environments in this way makes the translation slightly simpler than if we had used a Launchbury-style [16] call-by-need language with explicit environments. In the latter case, the translation would need to encode the environments. Here they are already encoded inside expressions. Answers are terminal computations: they do not reduce.

The first two reduction axioms (on the left) of the call-by-need semantics (Figure 5d) are obvious. The third axiom is the most important: it states that

$$\begin{array}{lrl}
\text{Types} & A, B & ::= \mathbf{bool} \mid A \to B \\
\text{Contexts} & \mathit{\Gamma} & ::= \diamond \mid \mathit{\Gamma}, x : A \\
\text{Expressions} & e & ::= x \mid \mathsf{diverge}_A \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \lambda x.\, e \mid e_1\, e_2
\end{array}$$

(a) Syntax

$$\frac{}{\mathit{\Gamma} \vdash x : A}\ \text{if } (x : A) \in \mathit{\Gamma} \qquad \frac{}{\mathit{\Gamma} \vdash \mathsf{diverge}_A : A}$$

$$\frac{}{\mathit{\Gamma} \vdash \mathsf{true} : \mathbf{bool}} \qquad \frac{}{\mathit{\Gamma} \vdash \mathsf{false} : \mathbf{bool}} \qquad \frac{\mathit{\Gamma} \vdash e_1 : \mathbf{bool} \qquad \mathit{\Gamma} \vdash e_2 : A \qquad \mathit{\Gamma} \vdash e_3 : A}{\mathit{\Gamma} \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : A}$$

$$\frac{\mathit{\Gamma}, x : A \vdash e : B}{\mathit{\Gamma} \vdash \lambda x.\, e : A \to B} \qquad \frac{\mathit{\Gamma} \vdash e_1 : A \to B \qquad \mathit{\Gamma} \vdash e_2 : A}{\mathit{\Gamma} \vdash e_1\, e_2 : B}$$

(b) Typing

$$\begin{array}{rcl}
\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 & \overset{\text{name}}{\rightsquigarrow} & e_2 \\
\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 & \overset{\text{name}}{\rightsquigarrow} & e_3 \\
(\lambda x.\, e)\, e' & \overset{\text{name}}{\rightsquigarrow} & e[x \mapsto e']
\end{array}
\qquad
\begin{array}{rcl}
\mathsf{diverge}_A & \overset{\text{name}}{\rightsquigarrow} & \mathsf{diverge}_A \\
\mathsf{if\ diverge}_{\mathbf{bool}}\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 & \overset{\text{name}}{\rightsquigarrow} & \mathsf{diverge}_A \\
\mathsf{diverge}_{A \to B}\, e' & \overset{\text{name}}{\rightsquigarrow} & \mathsf{diverge}_B
\end{array}$$

$$\frac{e_1 \overset{\text{name}}{\rightsquigarrow} e_1'}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \overset{\text{name}}{\rightsquigarrow} \mathsf{if}\ e_1'\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3} \qquad \frac{e_1 \overset{\text{name}}{\rightsquigarrow} e_1'}{e_1\, e_2 \overset{\text{name}}{\rightsquigarrow} e_1'\, e_2}$$

(c) Call-by-name operational semantics

$$\begin{array}{lrl}
\text{Evaluation contexts} & E[-] & ::= - \mid \mathsf{if}\ E[-]\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \\
 & & \quad \mid E[-]\, e_2 \mid (\lambda x.\, E[x])\, E'[-] \mid (\lambda x.\, E[-])\, e_2 \\
\text{Values} & v & ::= \mathsf{true} \mid \mathsf{false} \mid \lambda x.\, e \\
\text{Answers} & a & ::= v \mid (\lambda x.\, a)\, e
\end{array}$$

$$\begin{array}{rcl}
\mathsf{if\ true\ then}\ e_2\ \mathsf{else}\ e_3 & \overset{\text{need}}{\rightsquigarrow} & e_2 \\
\mathsf{if\ false\ then}\ e_2\ \mathsf{else}\ e_3 & \overset{\text{need}}{\rightsquigarrow} & e_3 \\
(\lambda x.\, E[x])\, v & \overset{\text{need}}{\rightsquigarrow} & (\lambda x.\, E[v])\, v \\
(\lambda x.\, a)\, e_1\, e_2 & \overset{\text{need}}{\rightsquigarrow} & (\lambda x.\, a\, e_2)\, e_1 \\
(\lambda x.\, E[x])\, ((\lambda y.\, a)\, e) & \overset{\text{need}}{\rightsquigarrow} & (\lambda y.\, (\lambda x.\, E[x])\, a)\, e
\end{array}
\qquad
\begin{array}{c}
\mathsf{diverge}_A \overset{\text{need}}{\rightsquigarrow} \mathsf{diverge}_A \\
E[\mathsf{diverge}_A] \overset{\text{need}}{\rightsquigarrow} \mathsf{diverge}_B \\
\dfrac{e \overset{\text{need}}{\rightsquigarrow} e'}{E[e] \overset{\text{need}}{\rightsquigarrow} E[e']}
\end{array}$$

(d) Call-by-need operational semantics

Fig. 5: The source language

if the subexpression currently being evaluated is a variable x, and the environment maps x to a source-language value v, then that use of x can be replaced with v. Note that $E[v]$ may contain other uses of x; the replacement only occurs when the value is actually needed. This axiom roughly corresponds to the first sequencing axiom of the equational theory of ECBPV (in Figure 3c). The fourth and fifth axioms of the call-by-need operational semantics rearrange the environment into a standard form. Both use a syntactic restriction to answers so that each expression has at most one reduct (this restriction is not needed to ensure that $\overset{\text{need}}{\rightsquigarrow}$ captures call-by-need). The rule on the right of the Figure 5d states that the reduction relation is a congruence (a needed subexpression can be reduced).

The two translations from the source language to ECBPV are given in Figure 6. The translation of types (Figure 6a) is shared between call-by-name and call-by-need. The two translations differ only for contexts and expressions. Types A are translated into value types $(\!|A|\!)$. The type **bool** becomes the two-element sum type $\mathbf{unit} + \mathbf{unit}$. The translation of a function type $A \to B$ is a thunked CBPV function type. The argument is a thunk of a computation that returns an $(\!|A|\!)$, and the result is a computation that returns a $(\!|B|\!)$.

For call-by-name (Figure 6b), contexts Γ are translated into contexts $(\!|\Gamma|\!)^{\text{name}}$ that contain thunks of computations. We could also have used contexts containing computation variables (omitting the thunks), but choose to use thunks to keep the translation as close as possible to previous translations into call-by-push-value. A well-typed expression $\Gamma \vdash e : A$ is translated into a ECBPV computation term $(\!|e|\!)^{\text{name}}$ that returns $(\!|A|\!)$, in context $(\!|\Gamma|\!)^{\text{name}}$. The translation of variables just forces the relevant variable in the context. The diverging computations $\mathsf{diverge}_A$ just use the diverging constants from our ECBPV signature. The translations of true and false are simple: they are computations that immediately return one of the elements of the sum type $\mathbf{unit} + \mathbf{unit}$. The translation of $\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$ first evaluates $(\!|e_1|\!)^{\text{name}}$, then uses the result to choose between $(\!|e_2|\!)^{\text{name}}$ and $(\!|e_3|\!)^{\text{name}}$. Lambdas are translated into computations that just return a thunked computation. Finally, application first evaluates the computation that returns a thunk of a function, and then forces this function, passing it a thunk of the argument.

For call-by-need (Figure 6c), contexts Γ are translated into contexts $(\!|\Gamma|\!)^{\text{need}}$, containing computations that return values. The computations in the context are all bound using need. An expression $\Gamma \vdash e : A$ is translated to a computation $(\!|e|\!)^{\text{need}}$ that returns $(\!|A|\!)$ in the context $(\!|\Gamma|\!)^{\text{need}}$. The typing is therefore similar to call-by-name. The key case is the translation of lambdas. These become computations that immediately return a thunk of a function. The function places the computation given as an argument onto the context using need, so that it is evaluated at most once, before executing the body. The remainder of the cases are similar to call-by-name.

Under the call-by-need translation, the expression $(\lambda x.\, e_1)\, e_2$ is translated into a term that executes the computation $(\!|e_1|\!)^{\text{need}}$, and executes $(\!|e_2|\!)^{\text{need}}$ only when needed. This is the case because, by the β rules for thunks, functions, and

$$(\!|\mathbf{bool}|\!) := \mathbf{unit} + \mathbf{unit} \qquad\qquad (\!|A \to B|\!) := \mathbf{U}\,(\mathbf{U}\,(\mathbf{Fr}\,(\!|A|\!)) \to \mathbf{Fr}\,(\!|B|\!))$$

(a) Translation $(\!|A|\!)$ of types

$\boxed{\text{Translation } (\!|\Gamma|\!)^{\mathrm{name}} \text{ of typing contexts}}$

$$(\!|\diamond|\!)^{\mathrm{name}} := \diamond \qquad\qquad (\!|\Gamma, x : A|\!)^{\mathrm{name}} := (\!|\Gamma|\!)^{\mathrm{name}}, x : \mathbf{U}\,(\mathbf{Fr}\,(\!|A|\!))$$

$\boxed{\text{Translation } (\!|\Gamma|\!)^{\mathrm{name}} \vdash (\!|e|\!)^{\mathrm{name}} : \mathbf{Fr}\,(\!|A|\!) \text{ of expressions}}$

$$\begin{aligned}
(\!|x|\!)^{\mathrm{name}} &:= \mathsf{force}\, x & (\!|\mathsf{diverge}_A|\!)^{\mathrm{name}} &:= \mathsf{force}\, \bot_A \\
(\!|\mathsf{true}|\!)^{\mathrm{name}} &:= \mathsf{return}\, \mathsf{inl}\, () & (\!|\mathsf{false}|\!)^{\mathrm{name}} &:= \mathsf{return}\, \mathsf{inr}\, () \\
(\!|\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3|\!)^{\mathrm{name}} &:= (\!|e_1|\!)^{\mathrm{name}}\ \mathsf{to}\ x.\ \mathsf{force}(\mathsf{case}\, x\, \mathsf{of} \\
&\qquad \{\mathsf{inl}\, z.\ \mathsf{thunk}(\!|e_2|\!)^{\mathrm{name}}, \mathsf{inr}\, z.\ \mathsf{thunk}(\!|e_3|\!)^{\mathrm{name}}\}) \\
(\!|\lambda x.\, e|\!)^{\mathrm{name}} &:= \mathsf{return}\, \mathsf{thunk}(\lambda x.\ (\!|e|\!)^{\mathrm{name}}) \\
(\!|e_1\, e_2|\!)^{\mathrm{name}} &:= (\!|e_1|\!)^{\mathrm{name}}\ \mathsf{to}\ z.\ (\mathsf{thunk}\ (\!|e_2|\!)^{\mathrm{name}})\ \text{‘}\ (\mathsf{force}\, z)
\end{aligned}$$

(b) Call-by-name translation

$\boxed{\text{Translation } (\!|\Gamma|\!)^{\mathrm{need}} \text{ of typing contexts}}$

$$(\!|\diamond|\!)^{\mathrm{need}} := \diamond \qquad\qquad (\!|\Gamma, x : A|\!)^{\mathrm{need}} := (\!|\Gamma|\!)^{\mathrm{need}}, \underline{x} : \mathbf{Fr}\,(\!|A|\!)$$

$\boxed{\text{Translation } (\!|\Gamma|\!)^{\mathrm{need}} \vdash (\!|e|\!)^{\mathrm{need}} : \mathbf{Fr}\,(\!|A|\!) \text{ of expressions}}$

$$\begin{aligned}
(\!|x|\!)^{\mathrm{need}} &:= \underline{x} & (\!|\mathsf{diverge}_A|\!)^{\mathrm{need}} &:= \mathsf{force}\, \bot_A \\
(\!|\mathsf{true}|\!)^{\mathrm{need}} &:= \mathsf{return}\, \mathsf{inl}\, () & (\!|\mathsf{false}|\!)^{\mathrm{need}} &:= \mathsf{return}\, \mathsf{inr}\, () \\
(\!|\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3|\!)^{\mathrm{need}} &:= (\!|e_1|\!)^{\mathrm{need}}\ \mathsf{to}\ x.\ \mathsf{force}(\mathsf{case}\, x\, \mathsf{of} \\
&\qquad \{\mathsf{inl}\, z.\ \mathsf{thunk}(\!|e_2|\!)^{\mathrm{need}}, \mathsf{inr}\, z.\ \mathsf{thunk}(\!|e_3|\!)^{\mathrm{need}}\}) \\
(\!|\lambda x.\, e|\!)^{\mathrm{need}} &:= \mathsf{return}\, \mathsf{thunk}(\lambda x'.\ (\mathsf{force}\, x')\ \mathsf{need}\ \underline{x}.\ (\!|e|\!)^{\mathrm{need}}) \\
(\!|e_1\, e_2|\!)^{\mathrm{need}} &:= (\!|e_1|\!)^{\mathrm{need}}\ \mathsf{to}\ z.\ (\mathsf{thunk}\ (\!|e_2|\!)^{\mathrm{need}})\ \text{‘}\ (\mathsf{force}\, z)
\end{aligned}$$

(c) Call-by-need translation

Fig. 6: Translation from the source language to ECBPV

returner types:

$$\llparenthesis(\lambda x.\, e_1)\, e_2\rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e_2\rrparenthesis^{\mathrm{need}}\ \mathsf{need}\ \underline{x}.\ \llparenthesis e_1\rrparenthesis^{\mathrm{need}}$$

As a consequence, translations of answers are particularly simple: they have the following form (up to $\equiv$):

$$M_1\ \mathsf{need}\ \underline{x}_1.\, M_2\ \mathsf{need}\ \underline{x}_2.\ \cdots M_n\ \mathsf{need}\ \underline{x}_n.\ \mathsf{return}\, V$$

which intuitively means the value V in the environment mapping each $\underline{x}_i$ to M_i.

It is easy to see that both translations produce terms with the correct types. We prove that both translations are *sound*: if $e \overset{\text{name}}{\leadsto} e'$ then $\llparenthesis e\rrparenthesis^{\mathrm{name}} \equiv \llparenthesis e'\rrparenthesis^{\mathrm{name}}$, and if $e \overset{\text{need}}{\leadsto} e'$ then $\llparenthesis e\rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e'\rrparenthesis^{\mathrm{need}}$. To do this for call-by-need, we first look at translations of evaluation contexts. The following lemma says the translation captures the idea that the hole in an evaluation context corresponds to the term being evaluated.

Lemma 4. *Define, for each evaluation context $E[-]$, the term $\mathcal{E}_y\llparenthesis E[-]\rrparenthesis^{\mathrm{need}}$ by:*

$$\begin{aligned}
\mathcal{E}_y\llparenthesis - \rrparenthesis^{\mathrm{need}} &\coloneqq \mathsf{return}\, y \\
\mathcal{E}_y\llparenthesis \mathsf{if}\ E[-]\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3\rrparenthesis^{\mathrm{need}} &\coloneqq \mathcal{E}\llparenthesis E[-]\rrparenthesis^{\mathrm{need}}\ \mathsf{to}\ x.\ \mathsf{force}(\mathsf{case}\, x\ \mathsf{of} \\
&\qquad\qquad \{\mathsf{inl}\, z.\, \mathsf{thunk}\llparenthesis e_2\rrparenthesis^{\mathrm{need}} \\
&\qquad\qquad ,\mathsf{inr}\, z.\, \mathsf{thunk}\llparenthesis e_3\rrparenthesis^{\mathrm{need}}\}) \\
\mathcal{E}_y\llparenthesis E[-]\, e_2\rrparenthesis^{\mathrm{need}} &\coloneqq \mathcal{E}_y\llparenthesis E[-]\rrparenthesis^{\mathrm{need}}\ \mathsf{to}\ z.\ \mathsf{thunk}\llparenthesis e_2\rrparenthesis^{\mathrm{need}}\ \text{‘}\ \mathsf{force}\, z \\
\mathcal{E}_y\llparenthesis (\lambda x.\, E[x])\, E'[-]\rrparenthesis^{\mathrm{need}} &\coloneqq \mathcal{E}_y\llparenthesis E'[-]\rrparenthesis^{\mathrm{need}}\ \mathsf{need}\ \underline{x}.\ \llparenthesis E[x]\rrparenthesis^{\mathrm{need}} \\
\mathcal{E}_y\llparenthesis (\lambda x.\, E[-])\, e_2\rrparenthesis^{\mathrm{need}} &\coloneqq \llparenthesis e_2\rrparenthesis^{\mathrm{need}}\ \mathsf{need}\ \underline{x}.\, \mathcal{E}_y\llparenthesis E[-]\rrparenthesis^{\mathrm{need}}
\end{aligned}$$

For each expression e we have:

$$\llparenthesis E[e]\rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e\rrparenthesis^{\mathrm{need}}\ \mathsf{to}\ y.\, \mathcal{E}_y\llparenthesis E[-]\rrparenthesis^{\mathrm{need}}$$

This lemma omits the typing of expressions for presentational purposes. It is easy to add suitable constraints on typing. Soundness is now easy to show:

Theorem 2 (Soundness). *For any two well-typed source-language expressions $\Gamma \vdash e : A$ and $\Gamma \vdash e' : A$:*

1. *If $e \overset{\text{name}}{\leadsto} e'$ then $\llparenthesis e\rrparenthesis^{\mathrm{name}} \equiv \llparenthesis e'\rrparenthesis^{\mathrm{name}}$.*
2. *If $e \overset{\text{need}}{\leadsto} e'$ then $\llparenthesis e\rrparenthesis^{\mathrm{need}} \equiv \llparenthesis e'\rrparenthesis^{\mathrm{need}}$.*

Now that we have sound call-by-name and call-by-need translations, we can state the meta-level equivalence formally. Suppose we are given a possibly open source-language expression $\Gamma \vdash e : B$. Recall that the call-by-need translation uses a context containing computation variables (i.e. $\llparenthesis\Gamma\rrparenthesis^{\mathrm{need}}$) and the call-by-name translation uses a context containing value variables, which map to thunks of computations. We have two ECBPV computation terms of type $\mathbf{Fr}\,\llparenthesis B\rrparenthesis$ in context $\llparenthesis\Gamma\rrparenthesis^{\mathrm{need}}$: one is just $\llparenthesis e\rrparenthesis^{\mathrm{need}}$, the other is $\llparenthesis e\rrparenthesis^{\mathrm{name}}$ with all of its variables substituted with thunked computations. The theorem then states that these are contextually equivalent.

Theorem 3 (Equivalence between call-by-name and call-by-need). *For all source-language expressions e satisfying $\underline{x}_1 : A_1, \ldots, \underline{x}_n : A_n \vdash e : B$*

$$(\!|e|\!)^{\text{name}}[x_1 \mapsto \mathsf{thunk}\,\underline{x}_1, \ldots, x_n \mapsto \mathsf{thunk}\,\underline{x}_n] \quad \cong_{\text{ctx}} \quad (\!|e|\!)^{\text{need}}$$

Proof. The proof of this theorem is by induction on the typing derivation of e. The interesting case is lambda abstraction, where we use the internal equivalence between call-by-name and call-by-need (Theorem 1).

4 An effect system for extended call-by-push-value

The equivalence between call-by-name and call-by-need in the previous section is predicated on the only effect in the language being nontermination. However, suppose the primitives of language have various effects (which means that in general the equivalence fails) but a given subprogram may be statically shown to have at most nontermination effects. In this case, we should be allowed to exploit the equivalence on the subprogram, interchanging call-by-need and call-by-name locally, even if the rest of the program uses other effects. In this section, we describe an *effect system* [20] for ECBPV, which statically estimates the side-effects of expressions, allowing us to exploit equivalences which hold only within subprograms. Effect systems can also be used for other purposes, such as proving the correctness of effect-dependent program transformations [29,7]. The ECBPV effect system also allows these.

Call-by-need makes statically estimating effects difficult. Computation variables bound using need might have effects on their first use, but on subsequent uses do not. Hence to precisely determine the effects of a term, we must track which variables have been used. McDermott and Mycroft [23] show how to achieve this for a call-by-need effect system; their technique can be adapted to ECBPV. Here we take a simpler approach. By slightly restricting the *effect algebras* we consider, we remove the need to track variable usage information, while still ensuring the effect information is not an underestimate (an underestimate would enable incorrect transformations). This can reduce the precision of the effect information obtained, but for our use case (determining equivalences between evaluation orders) this is not an issue, since we primarily care about which effects are used (rather than e.g. how many times they are used).

4.1 Effects

The effect system is parameterized by an *effect algebra*, which specifies the information that is tracked. Different effect algebras can be chosen for different applications. There are various forms of effect algebra. We follow Katsumata [15] and use *preordered monoids*, which are the most general.

Definition 5 (Preordered monoid). *A* preordered monoid $(\mathcal{F}, \leq, \cdot, 1)$ *consists of a monoid $(\mathcal{F}, \cdot, 1)$ and a preorder $\leq$ on $\mathcal{F}$, such that the binary operation $\cdot$ is monotone in each argument separately.*

Since we do not track variable usage information, we might misestimate the effect of a call-by-need computation variable evaluated for a second time (whose true effect is 1). To ensure this misestimate is an overestimate, we assume that the effect algebra is *pointed* (which is the case for most applications).

Definition 6 (Pointed preordered monoid). *A preordered monoid* $(\mathcal{F}, \leq, \cdot, 1)$ *is* pointed *if for all* $f \in \mathcal{F}$ *we have* $1 \leq f$*.*

The elements f of the set $\mathcal{F}$ are called *effects*. Each effect abstractly represents some potential side-effecting behaviours. The order $\leq$ provides *approximation* of effects. When $f \leq f'$ this means behaviours represented by f are included in those represented by f'. The binary operation $\cdot$ represents sequencing of effects, and 1 is the effect of a side-effect-free expression.

Traditional (*Gifford-style*) effect systems have some set Σ of *operations* (for example, $\Sigma \coloneqq \{\mathsf{read}, \mathsf{write}\}$), and use the preordered monoid $(\mathcal{P}\Sigma, \subseteq, \cup, \emptyset)$. In these cases, an effect f is just a set of operations. If a computation has effect f then f contains all of the operations the computation *may* perform. They can therefore be used to enforce that computations do not use particular operations. Another example is the preordered monoid $(\mathbb{N}^+, \leq, +, 1)$, which can be used to count the number of possible results a nondeterministic computation can return (or to count the number of times an operation is used).

In our example, where we wish to establish whether the effects of an expression are restricted to nontermination for our main example, we use the two-element preorder $\{\mathsf{diveff} \leq \top\}$ with join for sequencing and diveff as the unit 1. The effect diveff means side-effects restricted to (at most) nontermination, and $\top$ means unrestricted side-effects. Thus we would enable the equivalence between call-by-name and call-by-need when the effect is diveff, and not when it is $\top$. All of these examples are pointed. Others can be found in the literature.

4.2 Effect system and signature

The effect system includes effects within types. Specifically, each computation of returner type will have some side-effects when it is run, and hence each returner type $\mathbf{Fr}\, A$ is annotated with an element f of $\mathcal{F}$. We write the annotated type as $\langle f \rangle A$. Formally we replace the grammar of ECBPV computation types (and similarly, the grammar of typing contexts) with

$$\underline{C}, \underline{D} ::= \textstyle\prod_{i \in I} \underline{C}_i \mid A \to \underline{C} \mid \langle f \rangle A$$

$$\Gamma ::= \diamond \mid \Gamma, x : A \mid \Gamma, \underline{x} : \langle f \rangle A$$

(The highlighted parts indicate the differences.) The grammar used for value types is unchanged, except that it uses the new syntax of computation types.

The definition of ECBPV signature is similarly extended to contain the effect algebra as well as the set of constants:

Definition 7 (Signature). *A* signature $(\mathcal{F}, \mathcal{K})$ *consists of a pointed preordered monoid* $(\mathcal{F}, \leq, \cdot, 1)$ *of effects and, for each value type* A*, a set* $\mathcal{K}_A$ *of constants of type* A*, including* $() \in \mathcal{K}_{\mathbf{unit}}$*.*

$$\boxed{A <:_{\mathrm{v}} B}$$

$$\frac{}{\mathbf{unit} <:_{\mathrm{v}} \mathbf{unit}} \qquad \frac{A_1 <:_{\mathrm{v}} B_1 \qquad A_2 <:_{\mathrm{v}} B_2}{A_1 \times A_2 <:_{\mathrm{v}} B_1 \times B_2} \qquad \frac{A_1 <:_{\mathrm{v}} B_1 \qquad A_2 <:_{\mathrm{v}} B_2}{A_1 + A_2 <:_{\mathrm{v}} B_1 + B_2}$$

$$\frac{\underline{C} <: \underline{D}}{\mathbf{U}\,\underline{C} <:_{\mathrm{v}} \mathbf{U}\,\underline{D}}$$

$$\boxed{\underline{C} <: \underline{D}}$$

$$\frac{(\underline{C}_i <: \underline{D}_i)_{i \in I}}{\prod_{i \in I} \underline{C}_i <: \prod_{i \in I} \underline{D}_i} \qquad \frac{A <:_{\mathrm{v}} B \qquad \underline{C} <: \underline{D}}{(B \to \underline{C}) <: (A \to \underline{D})} \qquad \frac{A <:_{\mathrm{v}} B}{\langle f \rangle A <: \langle f' \rangle B} \text{ if } f \leq f'$$

Fig. 7: Subtyping in the ECBPV effect system

We assume a fixed effect system signature for the remainder of this section.

Since types contain effects, which have a notion of subeffecting, there is a natural notion of subtyping. We define (in Figure 7) two subtyping relations: $A <:_{\mathrm{v}} B$ for value types and $\underline{C} <: \underline{D}$ for computation types.

We treat the type constructor $\langle f \rangle$ as an operation on computation types by defining computation types $\langle f \rangle \underline{C}$.

$$\langle f \rangle \Big(\prod_{i \in I} \underline{C}_i \Big) \coloneqq \prod_{i \in I} \langle f \rangle \underline{C}_i \qquad \langle f \rangle (A \to \underline{C}) \coloneqq A \to \langle f \rangle \underline{C} \qquad \langle f \rangle (\langle f' \rangle A) \coloneqq \langle f \cdot f' \rangle A$$

This is an *action* of the preordered monoid on computation types. Its purpose is to give the typing rule for sequencing of computations. The sequencing of a computation with effect f with a computation of type $\underline{C}$ has type $\langle f \rangle \underline{C}$.

The typing judgements have exactly the same form as before (except for the new syntax of types). The majority of the typing rules, including all of the rules for value terms, are also unchanged. The only rules we change are those for computation variables, return, to and need, which are replaced with the first four rules in Figure 8. We also add two subtyping rules, one for values and one for computations. These are the last two rules of Figure 8.

The equational theory does not need to be changed to use it with the new effect system (except that the types appearing in each axiom now include effect information). For each axiom of the equational theory, the two terms still have the same type in the effect system. In particular, for the axiom

$$M \;\mathsf{need}\; \underline{x}.\, \underline{x} \;\mathsf{to}\; y.\, N \;\equiv\; M \;\mathsf{to}\; y.\, N[\underline{x} \mapsto \mathsf{return}\, y]$$

if $\Gamma \vdash M : \langle f \rangle A$ and $\Gamma, \underline{x} : \langle f \rangle A, y : A \vdash N : \underline{C}$ then the left-hand side has type $\langle f \rangle \underline{C}$. For the right-hand-side, we have $\Gamma, y : A \vdash N[\underline{x} \mapsto \mathsf{return}\, y] : \underline{C}$, because of the assumption that the preordered monoid is pointed (which implies $\mathsf{return}\, y$ can have *any* effect by subtyping, not just the unit effect 1). Hence the right-hand-side also has type $\langle f \rangle \underline{C}$. This axiom is the reason for our pointedness requirement.

$$\frac{}{\Gamma \vdash \underline{x} : \langle f \rangle A} \text{ if } (\underline{x} : \langle f \rangle A) \in \Gamma \qquad \frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Gamma \vdash \mathsf{return}\, V : \langle 1 \rangle A}$$

$$\frac{\Gamma \vdash M : \langle f \rangle A \qquad \Gamma, x : A \vdash N : \underline{C}}{\Gamma \vdash M \;\mathsf{to}\; x.\, N : \langle f \rangle \underline{C}} \qquad \frac{\Gamma \vdash M : \langle f \rangle A \qquad \Gamma, \underline{x} : \langle f \rangle A \vdash N : \underline{C}}{\Gamma \vdash M \;\mathsf{need}\; \underline{x}.\, N : \underline{C}}$$

$$\frac{\Gamma \vdash_{\mathrm{v}} V : A}{\Gamma \vdash_{\mathrm{v}} V : B} \text{ if } A <:_{\mathrm{v}} B \qquad \frac{\Gamma \vdash_{\mathrm{v}} M : \underline{C}}{\Gamma \vdash_{\mathrm{v}} N : \underline{D}} \text{ if } \underline{C} <: \underline{D}$$

Fig. 8: Effect system modifications to ECBPV

In particular, if we drop need from the language, the pointedness requirement is not required. Thus the rules we give also describe a fully general effect system for CBPV in which the effect algebra can be any preordered monoid.

4.3 Exploiting effect-dependent equivalences

Our primary goal in adding an effect system to ECBPV is to exploit (local, effect-justified) equivalences between evaluation orders even without a whole-language restriction on effects. We sketch how to do this for our example.

When proving the equivalence between call-by-name and call-by-need in Section 3 we assumed that the only constants in the language were () and $\bot_A : \mathbf{U}\,(\mathbf{Fr}\, A)$. To relax this restriction, we use the effect algebra with preorder $\{\mathsf{diveff} \leq \top\}$ described above, and change the type of $\bot_A$ from $\mathbf{U}\,(\mathbf{Fr}\, A)$ to $\mathbf{U}\,(\langle \mathsf{diveff} \rangle A)$. We can include other effectful constants, and give them the effect $\top$ (e.g. $\mathsf{write} : \mathbf{U}\,(V \to \langle \top \rangle \mathbf{unit})$).

The statement of the internal (object-level) equivalence becomes:

$$\text{if } \Gamma \vdash M : \langle \mathsf{diveff} \rangle A \text{ and } \Gamma, \underline{x} : \langle \mathsf{diveff} \rangle A \vdash N : \underline{C} \text{ then}$$
$$\Gamma \vdash M \;\mathsf{name}\; \underline{x}.\, N \cong_{\mathrm{ctx}} M \;\mathsf{need}\; \underline{x}.\, N : \underline{C}$$

The premise restricts the effect of M to diveff so that nontermination is its only possible side-effect. To prove this equivalence, we need a logical relation for the effect system, which means we have to define a Kripke relation $R_{\langle f \rangle A}$ for each effect f. For $R_{\langle \mathsf{diveff} \rangle A}$ we use the same definition as before (the definition of $R_{\mathbf{Fr} A}$). The definition of $R_{\langle \top \rangle A}$ depends on the specific other effects included.

To state and prove a meta-level equivalence for a source language that includes other side-effects, we need to define an effect system for the source language. This would use the same effect algebra as the ECBPV effect system, and be such that the translation of source language expressions preserves effects. To do this for the source language of Section 3, we replace the syntax of function types with $\langle f \rangle A \xrightarrow{f'} B$, where f is the effect of the argument (required due to lazy evaluation), and f' is the latent effect of the function (the effect it has after application). The translation is then

$$(\!|\langle f\rangle A \xrightarrow{f'} B|\!) \coloneqq \mathbf{U}\,(\mathbf{U}\,(\langle f\rangle (\!|A|\!)) \to \langle f'\rangle (\!|B|\!))$$

Just as for the object-level equivalence, the statement of the meta-level equivalence similarly requires the source-language expression to have the effect diveff. We omit the details here.

5 Related work

Metalanguages for evaluation order Call-by-push-value is similar to Moggi's monadic metalanguage [25], except for the distinction between computations and values. Both support several evaluation orders, but neither supports call-by-need. *Polarized* type theories [34] also take the approach of stratifying types into several kinds to capture multiple evaluation orders. Downen and Ariola [10] recently described how to capture call-by-need using polarity. They take a different approach to ours, by splitting up terms according to their evaluation order, rather than whether they might have effects. This means they have three kinds of type, resulting in a more complex language than ours. They also do not apply their language to reasoning about the differences between evaluation orders, which was the primary motivation for ECBPV. It is not clear whether their language can also be used for this purpose.

Multiple evaluation orders can also be captured in a Moggi-style language by using *joinads* instead of monads [28]. It is possible that there is some joinad structure implicit in extended call-by-push-value.

Reasoning about call-by-need The majority of work on reasoning about call-by-need source languages has concentrated on operational semantics based on environments [16], graphs [32,30], and answers [3,2,9,22]. However, these do not compare call-by-need with other evaluation orders. The only type-based analysis of a lazy source language we know of apart from McDermott and Mycroft's effect system [23] is [31,33].

Logical relations Kripke logical relations have previously been applied to the problems of lambda definability [12] and normalization [1,11]. Previous proofs of contextual equivalence relate only closed terms. We were forced to relate open terms because of the need construct.

Reasoning about effects using logical relations often runs into a difficulty in ensuring the relations are closed under sequencing of computations. We are able to work around this due to our specific choice of effects. It is possible that considering other effects would require a technique such as Lindley and Stark's *leapfrog method* [19,18].

Effect systems Effect systems have a long history, starting with Gifford-style effect systems [20]. We use preordered monoids as effect algebras following Katsumata [15]. Almost all of the previous work on effect systems has concentrated on call-by-value only. Kammar and Plotkin [14,13] describe a Gifford-style call-by-push-value effect system, though their formulation does not generalise to other effect algebras. Our effect system is the first general effect system for a CBPV-like language. The only previous work on call-by-need effects is [23].

There has also been much work on reasoning about program transformations using effect systems, e.g. [29,7,5,8,6,4]. We expect it to be possible to recast much of this in terms of extended call-by-push-value, and therefore apply these transformations for various evaluation orders.

6 Conclusions and future work

We have described extended call-by-push-value, a calculus that can be used for reasoning about several evaluation orders. In particular, ECBPV supports call-by-need via the addition of the construct $M \ \mathsf{need}\ \underline{x}.\, N$. This allows us to prove that call-by-name and call-by-need reduction are equivalent if nontermination is the only effect in the source language, both inside the language itself, and on the meta-level. We proved the latter by giving two translations of a source language into ECBPV: one that captures call-by-name reduction, and one that captures call-by-need reduction. We also defined an effect system for ECBPV. The effect system statically bounds the side-effects of terms, allowing equivalences between evaluation orders to be used without restricting the entire language to particular effects. We close with a description of possible future work.

Other equivalences between evaluation orders We have proved one example of an equivalence between evaluation orders using ECBPV, but there are others that we might also expect to hold. For example, we would expect call-by-need and call-by-value to be equivalent if the effects are restricted to nondeterminism, allocating state, and reading from state (but not writing). It should be possible to use ECBPV to prove these by defining suitable logical relations. More generally, it might be possible to characterize when particular equivalences hold in terms of the algebraic properties of the effects we restrict to.

Denotational semantics Using logical relations to prove contextual equivalence between terms directly is difficult. Adequate denotational semantics would allow us to reduce proofs of contextual equivalence to proofs of equalities in the model. Composing the denotational semantics with the call-by-need translation would also result in a call-by-need denotational semantics for the source language. Some potential approaches to describing the denotational semantics of ECBPV are Maraist et al.'s [21] translation into an affine calculus, combined with a semantics of linear logic [24], and also continuation-passing-style translations [27]. None of these consider side-effects however.

Acknowledgements

We gratefully acknowledge the support of an EPSRC studentship, and thank the anonymous reviewers for helpful comments.

References

1. Altenkirch, T., Hofmann, M., Streicher, T.: Categorical reconstruction of a reduction free normalization proof. Lecture Notes in Computer Science **953**, 182–199 (1995). https://doi.org/10.1007/3-540-60164-3_27
2. Ariola, Z.M., Felleisen, M.: The call-by-need lambda calculus. Journal of functional programming **7**(3), 265–301 (1997)
3. 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. pp. 233–246. ACM (1995). https://doi.org/10.1145/199448.199507
4. Benton, N., Hofmann, M., Nigam, V.: Effect-dependent transformations for concurrent programs. In: Proceedings of the 18th International Symposium on Principles and Practice of Declarative Programming. pp. 188–201. ACM (2016). https://doi.org/10.1145/2967973.2968602
5. Benton, N., Kennedy, A.: Monads, effects and transformations. Electronic Notes in Theoretical Computer Science **26**, 3–20 (1999). https://doi.org/10.1016/S1571-0661(05)80280-4
6. Benton, N., Kennedy, A., Hofmann, M., Nigam, V.: Counting successes: Effects and transformations for non-deterministic programs. In: Lindley, S., McBride, C., Trinder, P., Sannella, D. (eds.) A List of Successes That Can Change the World: Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday, pp. 56–72. Springer (2016). https://doi.org/10.1007/978-3-319-30936-1_3
7. Benton, N., Kennedy, A., Russell, G.: Compiling Standard ML to Java bytecodes. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. pp. 129–140. ACM (1998). https://doi.org/10.1145/289423.289435
8. Birkedal, L., Sieczkowski, F., Thamsborg, J.: A concurrent logical relation. In: Cégielski, P., Durand, A. (eds.) 21st EACSL Annual Conference on Computer Science Logic, CSL 2012. Leibniz International Proceedings in Informatics (LIPIcs), vol. 16, pp. 107–121. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2012). https://doi.org/10.4230/LIPIcs.CSL.2012.107
9. Chang, S., Felleisen, M.: The call-by-need lambda calculus, revisited. In: Proceedings of the 21st European Conference on Programming Languages and Systems. pp. 128–147. Springer-Verlag (2012). https://doi.org/10.1007/978-3-642-28869-2_7
10. Downen, P., Ariola, Z.M.: Beyond polarity: Towards a multi-discipline intermediate language with sharing. In: 27th EACSL Annual Conference on Computer Science Logic, CSL 2018. pp. 21:1–21:23 (2018). https://doi.org/10.4230/LIPIcs.CSL.2018.21
11. Fiore, M.: Semantic analysis of normalisation by evaluation for typed lambda calculus. In: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming. pp. 26–37. ACM (2002). https://doi.org/10.1145/571157.571161
12. Jung, A., Tiuryn, J.: A new characterization of lambda definability. In: Proceedings of the International Conference on Typed Lambda Calculi and Applications. pp. 245–257. Springer-Verlag, London, UK (1993). https://doi.org/10.1007/BFb0037110
13. Kammar, O.: Algebraic theory of type-and-effect systems. Ph.D. thesis, University of Edinburgh, UK (2014)

14. Kammar, O., Plotkin, G.D.: Algebraic foundations for effect-dependent optimisations. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 349–360. ACM (2012). https://doi.org/10.1145/2103656.2103698
15. Katsumata, S.: Parametric effect monads and semantics of effect systems. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 633–645. ACM (2014). https://doi.org/10.1145/2535838.2535846
16. Launchbury, J.: A natural semantics for lazy evaluation. In: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 144–154. ACM (1993). https://doi.org/10.1145/158511.158618
17. Levy, P.B.: Call-by-push-value: A subsuming paradigm. In: Girard, J.Y. (ed.) Typed Lambda Calculi and Applications. pp. 228–243. Springer, Berlin, Heidelberg (1999). https://doi.org/10.1007/3-540-48959-2_17
18. Lindley, S.: Normalisation by Evaluation in the Compilation of Typed Functional Programming Languages. Ph.D. thesis, University of Edinburgh, UK (2005)
19. Lindley, S., Stark, I.: Reducibility and $\top\top$-lifting for computation types. In: Proceedings of the 7th International Conference on Typed Lambda Calculi and Applications. pp. 262–277. Springer-Verlag, Berlin, Heidelberg (2005). https://doi.org/10.1007/11417170_20
20. Lucassen, J.M., Gifford, D.K.: Polymorphic effect systems. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 47–57. ACM (1988). https://doi.org/10.1145/73560.73564
21. 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. pp. 370–392 (1995). https://doi.org/10.1016/S0304-3975(98)00358-2
22. Maraist, J., Odersky, M., Wadler, P.: The call-by-need lambda calculus. Journal of Functional Programming **8**(3), 275–317 (1998). https://doi.org/10.1017/S0956796898003037
23. McDermott, D., Mycroft, A.: Call-by-need effects via coeffects. Open Computer Science **8**, 93–108 (2018). https://doi.org/10.1515/comp-2018-0009
24. Melliès, P.A.: Categorical semantics of linear logic. In: Interactive Models of Computation and Program Behaviour, Panoramas et Synthèses 27, Société Mathématique de France (2009)
25. Moggi, E.: Notions of computation and monads. Inf. Comput. **93**(1), 55–92 (1991). https://doi.org/10.1016/0890-5401(91)90052-4
26. Munch-Maccagnoni, G.: Models of a non-associative composition. In: Muscholl, A. (ed.) Foundations of Software Science and Computation Structures. pp. 396–410. Springer, Berlin, Heidelberg (2014)
27. Okasaki, C., Lee, P., Tarditi, D.: Call-by-need and continuation-passing style. LISP and Symbolic Computation **7**(1), 57–81 (1994). https://doi.org/10.1007/BF01019945
28. Petricek, T., Syme, D.: Joinads: A retargetable control-flow construct for reactive, parallel and concurrent programming. In: Proceedings of the 13th International Conference on Practical Aspects of Declarative Languages. pp. 205–219. Springer-Verlag, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18378-2_17
29. Tolmach, A.: Optimizing ML using a hierarchy of monadic types. In: Leroy, X., Ohori, A. (eds.) Types in Compilation. pp. 97–115. Springer, Berlin, Heidelberg (1998). https://doi.org/10.1007/BFb0055514

30. Turner, D.A.: A new implementation technique for applicative languages. Software: Practice and Experience **9**(1), 31–49 (1979). https://doi.org/10.1002/spe.4380090105
31. 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. pp. 1–11. ACM (1995). https://doi.org/10.1145/224164.224168
32. Wadsworth, C.: Semantics and Pragmatics of the Lambda-calculus. University of Oxford (1971)
33. Wansbrough, K., Peyton Jones, S.: Once upon a polymorphic type. In: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 15–28. ACM (1999). https://doi.org/10.1145/292540.292545
34. Zeilberger, N.: The Logical Basis of Evaluation Order and Pattern-matching. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2009)