Galois connecting call-by-value and call-by-name

Dylan McDermott

Reykjavik University, Iceland

Alan Mycroft

University of Cambridge, UK

Abstract

We establish a general framework for reasoning about the relationship between call-by-value and call-by-name.

In languages with side-effects, call-by-value and call-by-name executions of programs often have different, but related, observable behaviours. For example, if a program might diverge but otherwise has no side-effects, then whenever it terminates under call-by-value, it terminates with the same result under call-by-name. We propose a technique for stating and proving these properties. The key ingredient is Levy's call-by-push-value calculus, which we use as a framework for reasoning about evaluation orders. We construct maps between the call-by-value and call-by-name interpretations of types. We then identify properties of side-effects that imply these maps form a Galois connection. These properties hold for some side-effects (such as divergence), but not others (such as mutable state). This gives rise to a general reasoning principle that relates call-by-value and call-by-name. We apply the reasoning principle to example side-effects including divergence and nondeterminism.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning; Theory of computation → Denotational semantics; Theory of computation → Categorical semantics

Keywords and phrases computational effect, evaluation order, call-by-push-value, categorical semantics

Digital Object Identifier 10.4230/LIPIcs.FSCD.2022.29

Related Version *Extended Version*: `https://arxiv.org/abs/2202.08246` [16]

Funding *Dylan McDermott*: Supported by an EPSRC studentship, and by Icelandic Research Fund project grant no. 196323-053

1 Introduction

Suppose that we have a language in which terms can be statically tagged either as using call-by-value evaluation or as using call-by-name evaluation. Each program in this language would therefore use a mix of call-by-value and call-by-name at runtime. Given any such program M, we can construct a new program M' by changing call-by-value to call-by-name for some subterm. The question we consider in this paper is: what is the relationship between the observable behaviour of M and the observable behaviour of M'?

For a language with side-effects (such as divergence), changing the evaluation order in this way will in general change the behaviour of the program, but for some side-effects we can often say something about how we expect the behaviour to change:

- If there are no side-effects at all (in particular, programs are normalizing), the choice of evaluation order is irrelevant: M and M' terminate with the same result.
- If there are diverging terms (for instance, via recursion), then the behaviour may change: a program might diverge under call-by-value and return a result under call-by-name. However, we can say something about how the behaviour changes: if M terminates with some result, then M' terminates with the same result.
- If nondeterminism is the only side-effect, every result of M is a possible result of M'.

These three instances of the problem are intuitively obvious, and each can be proved separately. We develop a *general* technique for proving these properties.

The idea is to use a calculus that captures both call-by-value and call-by-name, as a setting in which we can reason about both evaluation orders (this is where M and M' live). The calculus we use is Levy's *call-by-push-value* (CBPV) [11]. Levy describes how to translate (possibly open) expressions e into CBPV terms $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$, which respectively correspond to call-by-value and call-by-name. We study the relationship between the behaviour of $(\!|e|\!)^{\mathrm{v}}$ and the behaviour of $(\!|e|\!)^{\mathrm{n}}$ in a given program context.

The main obstacle is that $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ have different types, and hence cannot be directly compared. Our solution to this is based on Reynolds's work relating direct and continuation semantics of the λ-calculus [25]: we identify maps between the call-by-value and call-by-name interpretations, and compose these with the translations of expressions to arrive at two terms that *can* be compared directly. We show that, under certain conditions (satisfied only for some side-effects, such as our examples), the maps between call-by-value and call-by-name form a *Galois connection* (Theorem 17). This fact gives rise to a general *reasoning principle* (Theorem 21) that we use to compare call-by-value with call-by-name. Given any preorder $\preccurlyeq$ that captures the property we wish to show about programs, our reasoning principle gives sufficiency conditions for showing $M \preccurlyeq M'$, where M' is constructed as above by replacing call-by-value with call-by-name. We apply our reasoning principle to examples by choosing different relations $\preccurlyeq$; each of these relations indicates the extent to which changing evaluation order affects the behaviour of the program. In the divergence example $N \preccurlyeq N'$ is defined to mean termination of N implies termination of N' with the same result; in the other examples $\preccurlyeq$ similarly mirrors the properties described informally above.

Rather than just considering some fixed collection of (allowable) side-effects, we work abstractly and identify properties of side-effects that enable us to relate call-by-value and call-by-name. An advantage of our approach is that the properties can be derived by looking at the structure of the two maps between evaluation orders.

Our reasoning principle relies on the existence of some denotational model of the side-effects. We construct the Galois connections and relate the call-by-value and call-by-name translations inside the model itself. Crucially, we use *order-enriched* models, which order the denotations of terms. The ordering on denotations is necessary to obtain a general reasoning principle. (Our example properties cannot be proved by showing that denotations are equal, because they are not symmetric.) Working inside the semantics rather than using syntactic logical relations makes it easier to prove and to use our reasoning principle, especially for the divergence example.

In Section 2 we summarize the call-by-push-value calculus (CBPV) and the call-by-value and call-by-name translations. We then make the following contributions:

- We describe an *order-enriched* categorical semantics for CBPV (Section 3).
- We define maps between the call-by-value and call-by-name translations (Section 4), and show that they form a Galois connection for side-effects satisfying certain conditions (Theorem 17).
- We use the Galois connection to prove a novel reasoning principle (Theorem 21) that relates the call-by-value and call-by-name translations of expressions (Section 5).

Throughout, we consider three different examples: no side-effects, divergence, and non-determinism. We apply our reasoning principle to each, proving all of the above properties. Our motivation is partly to demonstrate the Galois connection technique as a way of reasoning about different semantics of a given language. Call-by-value and call-by-name is one example of this (and Reynolds's original application to direct and continuation semantics is another).

2 Call-by-push-value, call-by-value, and call-by-name

Levy [11, 13] introduced call-by-push-value (CBPV) as a calculus that captures both call-by-value and call-by-name. We reason about the relationship between call-by-value and call-by-name evaluation inside CBPV.

The syntax of CBPV terms is stratified into two kinds: *values* V, W do not reduce, *computations* M, N might reduce (possibly with side-effects). The syntax of types is similarly stratified into *value types* A, B and *computation types* $\underline{C}, \underline{D}$.

$$
\begin{array}{rrcl}
\text{value types} & A, B & ::= & \mathbf{bool} \mid \mathbf{U}\,\underline{C} \\
\text{computation types} & \underline{C}, \underline{D} & ::= & A \to \underline{C} \mid \mathbf{F}\,A \\
\text{values} & V, W & ::= & x \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{thunk}\,M \\
\text{computations} & M, N & ::= & \lambda x{:}A.\,M \mid V`M \mid \mathbf{return}\,V \mid M\ \mathbf{to}\ x.\,N \\
& & & \mid \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 \mid \mathbf{force}\,V
\end{array}
$$

We include only a minimal subset of CBPV (containing higher-order functions, which are the main source of difficulty).

We include booleans (the value type **bool**) as a representative base type. The value type $\mathbf{U}\,\underline{C}$ is the type of *thunks* of computations of type $\underline{C}$. Elements of $\mathbf{U}\,\underline{C}$ are introduced using **thunk**: the value $\mathbf{thunk}\,M$ is the suspension of the computation term M. The corresponding eliminator is **force**, which is the inverse of **thunk**. Computation types include function types (where functions send values to computations). Function application is written $V`M$, where V is the argument and M is the function to apply. The *returner type* $\mathbf{F}\,A$ has as elements computations that return elements of the value type A; these computations may have side-effects. Elements of $\mathbf{F}\,A$ are introduced by **return**; the computation $\mathbf{return}\,V$ immediately returns the value V (with no side-effects). Computations can be sequenced using $M\ \mathbf{to}\ x.\,N$. This first evaluates M (which is required to have returner type), and then evaluates N with x bound to the result of M. (It is similar to `M >>= \x -> N` in Haskell.) The syntax we give here does not include any method of introducing effects; we extend CBPV with divergence (via recursion) and with nondeterminism in Section 2.2.

The evaluation order in CBPV is fixed for each program. The only primitive that causes the evaluation of two separate computations is **to**, which implements eager sequencing. Thunks give us more control over the evaluation order: they can be arbitrarily duplicated and discarded, and can be forced in any order chosen by the program. This is how CBPV captures both call-by-value and call-by-name.

CBPV has two typing judgments: $\Gamma \vdash V : A$ for values and $\Gamma \vdash_c M : \underline{C}$ for computations. *Typing contexts* Γ are ordered lists of (variable, value type) pairs. We require that no variable appears more than once in any typing context. Figure 1 gives the typing rules. Rules that add a new variable to a typing context implicitly require that the variable is fresh. We write $\diamond$ for the empty typing context, $V : A$ as an abbreviation for $\diamond \vdash V : A$, and $M : \underline{C}$ as an abbreviation for $\diamond \vdash_c M : \underline{C}$.

We give an operational semantics for CBPV. This consists of a big-step evaluation relation $M \Downarrow R$, which means the computation M evaluates to R. Here R ranges over *terminal computations*, which are the subset of computations with an introduction form on the outside:

$$R \ ::= \ \lambda x{:}A.\,M \mid \mathbf{return}\,V$$

We only evaluate closed, well-typed computations, so when we write $M \Downarrow R$ we assume $M : \underline{C}$ for some $\underline{C}$ (this implies $R : \underline{C}$). Reduction therefore cannot get stuck. The rules defining

$\boxed{\Gamma \vdash V : A}$

$$\frac{}{\Gamma \vdash x : A}\ \text{if } (x : A) \in \Gamma \qquad \frac{}{\Gamma \vdash \mathbf{true} : \mathbf{bool}} \qquad \frac{}{\Gamma \vdash \mathbf{false} : \mathbf{bool}} \qquad \frac{\Gamma \vdash_c M : \underline{C}}{\Gamma \vdash \mathbf{thunk}\, M : \mathbf{U}\,\underline{C}}$$

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

$$\frac{\Gamma, x : A \vdash_c M : \underline{C}}{\Gamma \vdash_c \lambda x{:}A.\, M : A \to \underline{C}} \qquad \frac{\Gamma \vdash V : A \quad \Gamma \vdash_c M : A \to \underline{C}}{\Gamma \vdash_c V\text{`}M : \underline{C}} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash_c \mathbf{return}\, V : \mathbf{F}\, A}$$

$$\frac{\Gamma \vdash_c M : \mathbf{F}\, A \quad \Gamma, x : A \vdash_c N : \underline{C}}{\Gamma \vdash_c M \ \mathbf{to}\ x.\, N : \underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{bool} \quad \Gamma \vdash_c M_1 : \underline{C} \quad \Gamma \vdash_c M_2 : \underline{C}}{\Gamma \vdash_c \mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2 : \underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{U}\,\underline{C}}{\Gamma \vdash_c \mathbf{force}\, V : \underline{C}}$$

Figure 1 CBPV typing rules

$$\frac{}{\lambda x{:}A.\, M \Downarrow \lambda x{:}A.\, M} \qquad \frac{M \Downarrow \lambda x{:}A.\, N \qquad N[x \mapsto V] \Downarrow R}{V\text{`}M \Downarrow R}$$

$$\frac{}{\mathbf{return}\, V \Downarrow \mathbf{return}\, V} \qquad \frac{M \Downarrow \mathbf{return}\, V \qquad N[x \mapsto V] \Downarrow R}{M\ \mathbf{to}\ x.\, N \Downarrow R} \qquad \frac{M \Downarrow R}{\mathbf{force}\,(\mathbf{thunk}\, M) \Downarrow R}$$

$$\frac{M_1 \Downarrow R}{\mathbf{if\ true\ then}\ M_1\ \mathbf{else}\ M_2 \Downarrow R} \qquad \frac{M_2 \Downarrow R}{\mathbf{if\ false\ then}\ M_1\ \mathbf{else}\ M_2 \Downarrow R}$$

Figure 2 Big-step operational semantics of CBPV

$\Downarrow$ are given in Figure 2. All terminal computations evaluate to themselves. Since we have not yet included any way of forming impure computations, the semantics is deterministic and normalizing: given any $M : \underline{C}$, there is exactly one terminal computation R such that $M \Downarrow R$. Section 2.2 extends the semantics in ways that violate these properties. We are primarily interested in evaluating computations of returner type.

A CBPV *program* is a closed computation $M : \mathbf{F}\,\mathbf{bool}$. The reasoning principle we give for call-by-value and call-by-name relates open terms in program contexts. A *program relation* consists of a preorder[1] $\preccurlyeq$ on closed computations of type $\mathbf{F}\,\mathbf{bool}$. For example, we could use

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V{:}\mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \ \Rightarrow\ (M' \Downarrow \mathbf{return}\, V)$$

We could also use, for example, the total relation for $\preccurlyeq$ (and in this case apply our reasoning principle for call-by-value and call-by-name even if we include e.g. mutable state as a side effect – but then of course the conclusion of our reasoning principle would be trivial). Given any program relation $\preccurlyeq$, we define a *contextual preorder* $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ on arbitrary well-typed computations (in typing context Γ) by considering the behaviour of M and M' in programs

[1] We do not actually need to assume that $\preccurlyeq$ is reflexive or transitive at any point, but because of constraints we add later (such as existence of an adequate model), it is unlikely that there are any interesting examples in which $\preccurlyeq$ is not a preorder.

$$\boxed{\tau \;\mapsto\; \text{value type } (\!|\tau|\!)^{\mathrm{v}}}$$

$$\begin{array}{rcl} \mathbf{bool} & \mapsto & \mathbf{bool} \\ \tau \to \tau' & \mapsto & \mathbf{U}\,((\!|\tau|\!)^{\mathrm{v}} \to \mathbf{F}\,(\!|\tau'|\!)^{\mathrm{v}}) \end{array}$$

$$\boxed{\Gamma \;\mapsto\; \text{typing context } (\!|\Gamma|\!)^{\mathrm{v}}}$$

$$\begin{array}{rcl} \diamond & \mapsto & \diamond \\ \Gamma, x : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{v}}, x : (\!|\tau|\!)^{\mathrm{v}} \end{array}$$

$$\boxed{\begin{array}{ccc} \text{expression} & & \text{computation} \\ \Gamma \vdash e : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{v}} \vdash_c (\!|e|\!)^{\mathrm{v}} : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \end{array}}$$

$$\begin{array}{rcl} x & \mapsto & \mathbf{return}\, x \\ \mathbf{true} & \mapsto & \mathbf{return}\,\mathbf{true} \\ \mathbf{false} & \mapsto & \mathbf{return}\,\mathbf{false} \\ \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & \mapsto & (\!|e_0|\!)^{\mathrm{v}}\ \mathbf{to}\ z.\,\mathbf{if}\ z\ \mathbf{then}\ (\!|e_1|\!)^{\mathrm{v}}\ \mathbf{else}\ (\!|e_2|\!)^{\mathrm{v}} \\ \lambda x{:}\tau.\,e & \mapsto & \mathbf{return}\,\mathbf{thunk}\,\lambda x{:}(\!|\tau|\!)^{\mathrm{v}}.\,(\!|e|\!)^{\mathrm{v}} \\ e\,e' & \mapsto & (\!|e|\!)^{\mathrm{v}}\ \mathbf{to}\ y.\,(\!|e'|\!)^{\mathrm{v}}\ \mathbf{to}\ z.\,z\,\text{`}\,\mathbf{force}\,y \end{array}$$

$$\boxed{\tau \;\mapsto\; \text{computation type } (\!|\tau|\!)^{\mathrm{n}}}$$

$$\begin{array}{rcl} \mathbf{bool} & \mapsto & \mathbf{F}\,\mathbf{bool} \\ \tau \to \tau' & \mapsto & (\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}) \to (\!|\tau'|\!)^{\mathrm{n}} \end{array}$$

$$\boxed{\Gamma \;\mapsto\; \text{typing context } (\!|\Gamma|\!)^{\mathrm{n}}}$$

$$\begin{array}{rcl} \diamond & \mapsto & \diamond \\ \Gamma, x : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{n}}, x : \mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}} \end{array}$$

$$\boxed{\begin{array}{ccc} \text{expression} & & \text{computation} \\ \Gamma \vdash e : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{n}} \vdash_c (\!|e|\!)^{\mathrm{n}} : (\!|\tau|\!)^{\mathrm{n}} \end{array}}$$

$$\begin{array}{rcl} x & \mapsto & \mathbf{force}\, x \\ \mathbf{true} & \mapsto & \mathbf{return}\,\mathbf{true} \\ \mathbf{false} & \mapsto & \mathbf{return}\,\mathbf{false} \\ \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 & \mapsto & (\!|e_0|\!)^{\mathrm{n}}\ \mathbf{to}\ z.\,\mathbf{if}\ z\ \mathbf{then}\ (\!|e_1|\!)^{\mathrm{n}}\ \mathbf{else}\ (\!|e_2|\!)^{\mathrm{n}} \\ \lambda x{:}\tau.\,e & \mapsto & \lambda x{:}\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}.\,(\!|e|\!)^{\mathrm{n}} \\ e\,e' & \mapsto & (\mathbf{thunk}\,(\!|e'|\!)^{\mathrm{n}})\,\text{`}\,(\!|e|\!)^{\mathrm{n}} \end{array}$$

Figure 3 Call-by-value (left) and call-by-name (right) translations into CBPV

as follows. A *computation context* $\mathcal{E}$ is a computation term, with a single hole $\square$ where a computation term is expected. We write $\mathcal{E}[M]$ for the computation that results from replacing $\square$ with M (which may capture some of the free variables of M).

▶ **Definition 1** (Contextual preorder). *Suppose that $\preccurlyeq$ is a program relation, and that $\Gamma \vdash_c M : \underline{C}$ and $\Gamma \vdash_c M' : \underline{C}$ are two computations of the same type. We write $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ if, for all computation contexts $\mathcal{E}$ such that $\mathcal{E}[M], \mathcal{E}[M'] : \mathbf{F}\,\mathbf{bool}$, we have $\mathcal{E}[M] \preccurlyeq \mathcal{E}[M']$. We write $M \cong^{\Gamma}_{\mathrm{ctx}} M'$, and say that M and M' are* contextually equivalent*, when both $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ and $M' \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M$ hold.*

We sometimes omit Γ, and write just $M \preccurlyeq_{\mathrm{ctx}} M'$ or $M \cong_{\mathrm{ctx}} M'$.

2.1 Call-by-value and call-by-name

We use CBPV (instead of e.g. the monadic metalanguage [20]) because it captures both call-by-value and call-by-name evaluation. Levy [11] gives two compositional translations from a source language into CBPV: one for call-by-value and one for call-by-name. We recall both translations in this section; our goal is to reason about the relationship between them.

For the source language, we use the following syntax of types τ and expressions e:

$$\tau \ ::=\ \mathbf{bool} \mid \tau \to \tau' \qquad e \ ::=\ x \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ e_0\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \mid \lambda x{:}\tau.\,e \mid e\,e'$$

The source language has a typing judgement of the form $\Gamma \vdash e : \tau$, defined by the usual rules.

The two translations from the source language to CBPV are defined in Figure 3. For call-by-value, each source language type τ is mapped to a CBPV value type $(\!|\tau|\!)^{\mathrm{v}}$ that contains the results of call-by-value computations. For call-by-name, τ is translated to a computation type $(\!|\tau|\!)^{\mathrm{n}}$, which contains the computations themselves. Functions under the call-by-value translation accept values of type $(\!|\tau|\!)^{\mathrm{v}}$ as arguments; arguments are evaluated before being passed to the function. Under the call-by-name translation, functions accept thunks of computations as arguments; instead of evaluating them, arguments are thunked before passing them to call-by-name functions. Source-language typing contexts Γ are translated to CBPV typing contexts $(\!|\Gamma|\!)^{\mathrm{v}}$ and $(\!|\Gamma|\!)^{\mathrm{n}}$. In call-by-value they contain values, in call-by-name they contain thunks of computations. Source-language expressions e are mapped to CBPV computations $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$. The translation uses some auxiliary program variables, which are assumed fresh. Levy [11] proves that, in a precise sense, these translations do indeed capture call-by-value and call-by-name.

2.2 Examples

We consider three collections of (allowable) side-effects as examples throughout the paper.

▶ **Example 2** (No side-effects). We include the simplest possible example: the case where there are no side-effects at all. For this example, call-by-value and call-by-name turn out to have identical behaviour. We define the program relation $M \preccurlyeq M'$ (for closed computations $M, M' : \mathbf{F}\,\mathbf{bool}$) as:

$$M \preccurlyeq M' \quad \text{if and only if} \quad \exists V\!:\!\mathbf{bool}.\ (M \Downarrow \mathbf{return}\,V) \ \wedge\ (M' \Downarrow \mathbf{return}\,V)$$

In other words, M and M' both evaluate to the same result V. The contextual preorder $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ means if we construct two programs by wrapping M and M' in the same computation context, then these two programs evaluate to the same result. This relation is symmetric. Our other examples use non-symmetric relations.

▶ **Example 3** (Divergence). For our second example, the only side-effect is divergence (via recursion). In this case, call-by-value and call-by-name do not have identical behaviour (they are not related by $\preccurlyeq_{\mathrm{ctx}}$ as it is defined in our no-side-effects example). We instead show that replacing call-by-value with call-by-name does not change a terminating program into a diverging one.

We extend our two languages with recursion. For CBPV we extend the syntax of computations with fixed points $\mathbf{rec}\,x\!:\!\mathbf{U}\underline{C}.\,M$, and correspondingly extend the type system and operational semantics with the following rules:

$$\frac{\Gamma, x : \mathbf{U}\underline{C} \vdash_c M \ :\ \underline{C}}{\Gamma \vdash_c \mathbf{rec}\,x\!:\!\mathbf{U}\underline{C}.\,M \ :\ \underline{C}} \qquad \frac{M[x \mapsto \mathbf{thunk}\,(\mathbf{rec}\,x\!:\!\mathbf{U}\underline{C}.\,M)] \Downarrow R}{\mathbf{rec}\,x\!:\!\mathbf{U}\underline{C}.\,M \Downarrow R}$$

Of course, by adding recursion we lose normalization (but the semantics is still deterministic). We extend the source language, and the two translations into CBPV, with recursive functions:

$$e \ ::=\ \ldots \mid \mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e \qquad \frac{\Gamma, f : \tau \to \tau', x : \tau \vdash e : \tau'}{\Gamma \vdash \mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e : \tau \to \tau'}$$

$$\begin{aligned}
(\!|\mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e|\!)^{\mathrm{v}} &= \mathbf{return}\,\mathbf{thunk}\,(\mathbf{rec}\,f\!:\!\mathbf{U}\,((\!|\tau|\!)^{\mathrm{v}} \to \mathbf{F}\,(\!|\tau'|\!)^{\mathrm{v}}).\,\lambda x\!:\!(\!|\tau|\!)^{\mathrm{v}}.\,(\!|e|\!)^{\mathrm{v}})\\
(\!|\mathbf{rec}\,f\!:\!\tau \to \tau'.\,\lambda x.\,e|\!)^{\mathrm{n}} &= \mathbf{rec}\,f\!:\!\mathbf{U}\,(\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}} \to (\!|\tau'|\!)^{\mathrm{n}}).\,\lambda x\!:\!\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}.\,(\!|e|\!)^{\mathrm{n}}
\end{aligned}$$

The expression $\Omega_\tau = ((\mathbf{rec}\,f\!:\!\mathbf{bool} \to \tau.\,\lambda x.\,f\,x)\,\mathbf{false}) : \tau$ enables us to distinguish between call-by-value and call-by-name: $(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau$ diverges in call-by-value but not in call-by-name. In particular, we have $(\!|(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,\mathbf{true}$, but there is no R such that $(\!|(\lambda x\!:\!\tau.\,\mathbf{true})\,\Omega_\tau|\!)^{\mathrm{v}} \Downarrow R$.

For this example, we define the program relation $\preccurlyeq$ by

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V : \mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \;\Rightarrow\; (M' \Downarrow \mathbf{return}\, V)$$

so that $M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$ informally means if a program containing M terminates with some result then the same program with M' instead of M terminates with the same result.

▶ **Example 4** (Nondeterminism)**.** Finally, we consider finite nondeterminism. Again call-by-value and call-by-name have different behaviour, but any result of a call-by-value execution is also a result of a call-by-name execution (if suitable nondeterministic choices are made).

We consider CBPV without recursion, but augmented with computations $\mathbf{fail}_{\underline{C}}$ for nullary nondeterministic choice and $M\, \mathbf{or}\, N$ for binary nondeterministic choice between computations; the typing and evaluation rules are standard:

$$\frac{}{\Gamma \vdash_c \mathbf{fail}_{\underline{C}} : \underline{C}} \qquad \frac{\Gamma \vdash_c M : \underline{C} \qquad \Gamma \vdash_c N : \underline{C}}{\Gamma \vdash_c M \;\mathbf{or}\; N : \underline{C}} \qquad \frac{M \Downarrow R}{M \;\mathbf{or}\; N \Downarrow R} \qquad \frac{N \Downarrow R}{M \;\mathbf{or}\; N \Downarrow R}$$

(There is no R such that $\mathbf{fail}_{\underline{C}} \Downarrow R$.) We similarly include nullary and binary nondeterminism in the source language, and extend the call-by-value and call-by-name translations:

$$\frac{}{\Gamma \vdash \mathbf{fail}_{\tau} : \tau} \qquad \frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash e \;\mathbf{or}\; e' : \tau} \qquad \begin{array}{c} (\!|\mathbf{fail}_{\tau}|\!)^{\mathrm{v}} = \mathbf{fail}_{\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}} \\ (\!|e \;\mathbf{or}\; e'|\!)^{\mathrm{v}} = (\!|e|\!)^{\mathrm{v}} \;\mathbf{or}\; (\!|e'|\!)^{\mathrm{v}} \end{array} \qquad \begin{array}{c} (\!|\mathbf{fail}_{\tau}|\!)^{\mathrm{n}} = \mathbf{fail}_{(\!|\tau|\!)^{\mathrm{n}}} \\ (\!|e \;\mathbf{or}\; e'|\!)^{\mathrm{n}} = (\!|e|\!)^{\mathrm{n}} \;\mathbf{or}\; (\!|e'|\!)^{\mathrm{n}} \end{array}$$

As an example, evaluating the expression $e = (\lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ x\ \mathbf{else\ true})(\mathbf{true\ or\ false})$ under call-by-value necessarily results in **true**, but under call-by-name we can also get **false**. (We have $(\!|e|\!)^{\mathrm{v}} \not\Downarrow \mathbf{return\,false}$ but $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return\,false}$.)

For nondeterminism, we define $\preccurlyeq$ in the same way as our divergence example:

$$M \preccurlyeq M' \quad \text{if and only if} \quad \forall V : \mathbf{bool}.\ (M \Downarrow \mathbf{return}\, V) \;\Rightarrow\; (M' \Downarrow \mathbf{return}\, V)$$

This captures the property that any result that arises from an execution of M (which may involve call-by-value) might arise from an execution of M' (which may involve call-by-name).

3 Order-enriched denotational semantics

We give a denotational semantics for CBPV, which we use to prove instances of $\preccurlyeq_{\mathrm{ctx}}$. Since $\preccurlyeq_{\mathrm{ctx}}$ is not in general symmetric, we use *order-enriched* models, which come with partial orders $\sqsubseteq$ between denotations. In an *adequate* model, $[\![M]\!] \sqsubseteq [\![N]\!]$ implies $M \preccurlyeq_{\mathrm{ctx}} N$. Our semantics is based on Levy's *algebra models* [13] for CBPV, in which each computation type is interpreted as a monad algebra. (We restrict to algebra models for simplicity. Other forms of model, such as *adjunction models* [12] can be used for the same purpose.)

We assume no knowledge of enriched category theory; instead we give the relevant order-enriched (specifically **Poset**-enriched) definitions here. (We do however assume some basic ordinary category theory.)

▶ **Definition 5.** *A* **Poset**-category $\mathbf{C}$ *is an ordinary category, together with a partial order* $\sqsubseteq$ *on each hom-set* $\mathbf{C}(X, Y)$*, such that composition is monotone.*

If $\mathbf{C}$ is a **Poset**-category, we refer to the ordinary category as the *underlying* ordinary category, and write $|\mathbf{C}|$ for the class of objects.

▶ **Example 6.** We use the following three **Poset**-categories.

Poset-category $\mathbf{C}$	Objects $X \in |\mathbf{C}|$	Morphisms $f : X \to Y$	Order $f \sqsubseteq f'$
Set	sets	functions	equality
Poset	posets	monotone functions	pointwise
ω**Cpo**	ωcpos	ω-continuous functions	pointwise

In each case, composition and identities are defined in the usual way. For **Set**, since the hom-posets $\mathbf{Set}(X, Y)$ are discrete, all of the **Poset**-enriched definitions coincide with the ordinary (unenriched) definitions. The objects of $\omega\mathbf{Cpo}$ are posets $(X, \sqsubseteq)$ for which $\sqsubseteq$ is ω-*complete*, i.e. for which every ω-chain $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ has a least upper bound $\bigsqcup x$. Morphisms are ω-*continuous* functions, i.e. monotone functions that preserve least upper bounds of ω-chains.

Let $\mathbf{C}$ be a **Poset**-category. We say that $\mathbf{C}$ is *cartesian* when its underlying category has a terminal object 1 and binary products $X_1 \times X_2$, such that the pairing functions $\langle -, - \rangle : \mathbf{C}(W, X_1) \times \mathbf{C}(W, X_2) \to \mathbf{C}(W, X_1 \times X_2)$ are monotone. When this is the case, there are canonical isomorphisms $assoc_{X_1,X_2,X_3} : (X_1 \times X_2) \times X_3 \to X_1 \times (X_2 \times X_3)$. We say that $\mathbf{C}$ is *cartesian closed* when it is cartesian and its underlying category has exponentials $X \Rightarrow Y$ for which the currying functions $\Lambda : \mathbf{C}(W \times X, Y) \to \mathbf{C}(W, X \Rightarrow Y)$ are monotone. (It follows that the uncurrying functions $\Lambda^{-1} : \mathbf{C}(W, X \Rightarrow Y) \to \mathbf{C}(W \times X, Y)$ are also monotone.) We write $ev_{X,Y}$ for the canonical morphism $\Lambda^{-1} id : (X \Rightarrow Y) \times X \to Y$. Binary *coproducts* in $\mathbf{C}$ are just binary coproducts in the underlying ordinary category, except that the copairing functions $[-, -] : \mathbf{C}(X_1, W) \times \mathbf{C}(X_2, W) \to \mathbf{C}(X_1 + X_2, W)$ are required to be monotone. The **Poset**-categories **Set**, **Poset**, and ω**Cpo** are all cartesian closed, and have binary coproducts.

We interpret computation types as (Eilenberg-Moore) algebras for an order-enriched monad T, which we need to be *strong* (just as models of Moggi's monadic metalanguage [20] use a strong monad). The definitions of strong **Poset**-monad and of T-algebra we give are slightly non-standard, but are equivalent to the standard ones (see for example [17]). (In particular, it is more convenient for us to bake the strength into the Kleisli extension of the monad instead of having a separate strength.)

▶ **Definition 7** (Strong **Poset**-monad)**.** *A* strong **Poset**-monad $\mathsf{T} = (T, \eta, (-)^{\dagger})$ *on a cartesian* **Poset**-*category* $\mathbf{C}$ *consists of an object* $TX \in |\mathbf{C}|$ *and morphism* $\eta_X : X \to TX$ *for each* $X \in |\mathbf{C}|$*, and a monotone function (*Kleisli extension*)* $(-)^{\dagger} : \mathbf{C}(W \times X, TY) \to \mathbf{C}(W \times TX, TY)$ *for each* $W, X, Y \in |\mathbf{C}|$*, such that*

$$\begin{aligned}
f^{\dagger} \circ (id_W \times \eta_X) &= f : W \times X \to TY && \textit{for all } f : W \times X \to TY \\
(\eta_X \circ \pi_2)^{\dagger} &= \pi_2 : 1 \times TX \to TX && \textit{for all } X \in |\mathbf{C}| \\
(g^{\dagger} \circ (id_{W'} \times f) \circ assoc)^{\dagger} &= g^{\dagger} \circ (id_{W'} \times f^{\dagger}) \circ assoc && \textit{for all } f : W \times X \to TY, \\
&\quad : (W' \times W) \times TX \to TZ && \qquad g : W' \times Y \to TZ
\end{aligned}$$

Specializing the Kleisli extension of T to $W = 1$ produces a (non-strong) extension operator $(-)^{\dagger} : \mathbf{C}(X, TY) \to \mathbf{C}(TX, TY)$. We use this to define, for every $f : X \to Y$, a morphism $Tf : TX \to TY$ by $Tf = (\eta_Y \circ f)^{\dagger}$. (The latter definition makes T into a **Poset**-*functor*.)

▶ **Definition 8** (Eilenberg-Moore algebra)**.** *Let* T *be a strong* **Poset**-*monad on a cartesian* **Poset**-*category* $\mathbf{C}$*. A* T-algebra $\mathsf{Z} = (Z, (-)^{\ddagger})$ *is a pair of an object* $Z \in |\mathbf{C}|$ *(the* carrier*) and monotone function (*extension *operator)* $(-)^{\ddagger} : \mathbf{C}(W \times X, Z) \to \mathbf{C}(W \times TX, Z)$ *for each*

$W, X \in |\mathbf{C}|$, *such that*

$$f^{\ddagger} \circ (id_W \times \eta_X) = f : W \times X \to Z \qquad \textit{for all } f : W \times X \to TY$$

$$(g^{\ddagger} \circ (id_{W'} \times f) \circ assoc)^{\ddagger} = g^{\ddagger} \circ (id_{W'} \times f^{\dagger}) \circ assoc \qquad \textit{for all } f : W \times X \to TY,$$
$$: (W' \times W) \times TX \to Z \qquad g : W' \times Y \to Z$$

For each $X \in |\mathbf{C}|$*, we write* $F_{\mathsf{T}}X$ *for the* free T-algebra $(TX, (-)^{\dagger})$*, and for each* T*-algebra* Z*, we write* $U_{\mathsf{T}}\mathsf{Z}$ *for the carrier* $Z \in |\mathbf{C}|$.

Specializing the extension operator of a T-algebra Z to $W = 1$ produces a (non-strong) extension operator $(-)^{\ddagger} : \mathbf{C}(X, Z) \to \mathbf{C}(TX, Z)$.

Let T be a strong **Poset**-monad on a cartesian closed **Poset**-category $\mathbf{C}$. If $Y \in |\mathbf{C}|$ and Z is a T-algebra, then there is a T-algebra $Y \Rightarrow \mathsf{Z}$ with carrier $Y \Rightarrow Z$ and extension operator

$$f^{\ddagger} = \Lambda((\Lambda^{-1} f \circ \beta_{W,Y,X})^{\ddagger} \circ \beta_{W,TX,Y}) : W \times TX \to Y \Rightarrow Z \qquad \text{for } f : W \times X \to Y \Rightarrow Z$$

where $\beta_{X_1,X_2,X_3} = \langle\langle \pi_1 \circ \pi_1, \pi_2\rangle, \pi_2 \circ \pi_1\rangle : (X_1 \times X_2) \times X_3 \to (X_1 \times X_3) \times X_2$. These provide the interpretations of function types $A \to \underline{C}$.

▶ **Definition 9.** *A* model *of CBPV consists of*

- *a cartesian closed* **Poset***-category* $\mathbf{C}$*;*
- *the coproduct* $2 = 1 + 1$*, together with the corresponding morphisms* $inl, inr : 1 \to 2$*;*
- *a strong* **Poset***-monad* $\mathsf{T} = (T, \eta, (-)^{\dagger})$ *on* $\mathbf{C}$.

Given any model, the interpretation $[\![-]\!]$ of CBPV is defined in Figure 4. Value types A are interpreted as objects $[\![A]\!] \in |\mathbf{C}|$, while computation types $\underline{C}$ are interpreted as T-algebras. Typing contexts Γ are interpreted as objects $[\![\Gamma]\!] \in \mathbf{C}$ using the cartesian structure of $\mathbf{C}$; if $(x : A) \in \Gamma$ then we write π_x for the corresponding projection $[\![\Gamma]\!] \to [\![A]\!]$. Values $\Gamma \vdash V : A$ (respectively computations $\Gamma \vdash_c M : \underline{C}$) are interpreted as morphisms $[\![\Gamma \vdash V : A]\!]$ (resp. $[\![\Gamma \vdash_c M : \underline{C}]\!]$) in $\mathbf{C}$; we often omit the typing context and type when writing these. Programs $\diamond \vdash_c M : \mathbf{bool}$ are therefore interpreted as morphisms $[\![M]\!] : 1 \to T2$. To interpret **if**, we use the fact that, since $\mathbf{C}$ is cartesian closed, products distribute over the coproduct $2 = 1 + 1$. This means that for every $W \in |\mathbf{C}|$, the coproduct $W + W$ also exists in $\mathbf{C}$, and the canonical morphism

$$W + W \xrightarrow{[\langle id_W, inl \circ \langle\rangle_W\rangle, \langle id_W, inr \circ \langle\rangle_W\rangle]} W \times 2$$

has an inverse $dist_W : W \times 2 \to W + W$.

By composing the semantics of CBPV with the two translations of the source language, we obtain a call-by-value semantics $[\![-]\!]^{\mathrm{v}} = [\![(\!|-|\!)^{\mathrm{v}}]\!]$ and a call-by-name semantics $[\![-]\!]^{\mathrm{n}} = [\![(\!|-|\!)^{\mathrm{n}}]\!]$ of the source language.

We use the denotational semantics as a tool for proving instances of contextual preorders; for this we need *adequacy*.

▶ **Definition 10.** *A model of CBPV is* adequate *(with respect to a given program relation* $\preccurlyeq$*) if for all computations* $\Gamma \vdash_c M : \underline{C}$ *and* $\Gamma \vdash_c M' : \underline{C}$ *we have*

$$[\![\Gamma \vdash_c M : \underline{C}]\!] \sqsubseteq [\![\Gamma \vdash_c M' : \underline{C}]\!] \quad \Rightarrow \quad M \preccurlyeq^{\Gamma}_{\mathrm{ctx}} M'$$

We give three different models, one for each of our three examples in Section 2.2. Each model is adequate with respect to the corresponding definition of $\preccurlyeq$; the proof in each case is a standard *logical relations* argument (e.g. [31]).

$\boxed{\mathbf{C}\text{-object } [\![A]\!]}$

$$\begin{aligned}[\![\mathbf{bool}]\!] &= 2 \quad (= 1{+}1)\\ [\![\mathbf{U}\,\underline{C}]\!] &= U_{\mathsf{T}}[\![\underline{C}]\!]\end{aligned}$$

$\boxed{\mathsf{T}\text{-algebra } [\![\underline{C}]\!]}$

$$\begin{aligned}[\![A \to \underline{C}]\!] &= [\![A]\!] \Rightarrow [\![\underline{C}]\!]\\ [\![\mathbf{F}\,A]\!] &= F_{\mathsf{T}}[\![A]\!]\end{aligned}$$

$\boxed{\mathbf{C}\text{-object } [\![\Gamma]\!]}$

$$\begin{aligned}[\![\diamond]\!] &= 1\\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!]\end{aligned}$$

$\boxed{[\![\Gamma \vdash V : A]\!] : [\![\Gamma]\!] \to [\![A]\!]}$

$$\begin{aligned}[\![x]\!] &= \pi_x\\ [\![\mathbf{true}]\!] &= \mathit{inl} \circ \langle\rangle_{[\![\Gamma]\!]}\\ [\![\mathbf{false}]\!] &= \mathit{inr} \circ \langle\rangle_{[\![\Gamma]\!]}\\ [\![\mathbf{thunk}\,M]\!] &= [\![M]\!]\end{aligned}$$

$\boxed{[\![\Gamma \vdash_c M : \underline{C}]\!] : [\![\Gamma]\!] \to U_{\mathsf{T}}[\![\underline{C}]\!]}$

$$\begin{aligned}[\![\lambda x{:}A.\,M]\!] &= \Lambda[\![M]\!]\\ [\![V\text{`}M]\!] &= \Lambda^{-1}[\![M]\!] \circ \langle \mathit{id}, [\![V]\!]\rangle\\ [\![\mathbf{return}\,V]\!] &= \eta \circ [\![V]\!]\\ [\![M \textbf{ to } x.\,N]\!] &= [\![N]\!]^{\ddagger} \circ \langle \mathit{id}, [\![M]\!]\rangle\\ [\![\mathbf{if}\ V\ \mathbf{then}\ M_1\ \mathbf{else}\ M_2]\!] &= [[\![M_1]\!], [\![M_2]\!]] \circ \mathit{dist} \circ \langle \mathit{id}, [\![V]\!]\rangle\\ [\![\mathbf{force}\,V]\!] &= [\![V]\!]\end{aligned}$$

Figure 4 Denotational semantics of CBPV

▶ **Example 11.** For CBPV with no side-effects, we use $\mathbf{C} = \mathbf{Set}$. The strong $\mathbf{Poset}$-monad T is the identity on $\mathbf{Set}$. Each T-algebra Z is completely determined by its carrier Z; the extension operator $(-)^{\ddagger} : \mathbf{Set}(W \times X, Z) \to \mathbf{Set}(W \times X, Z)$ is necessarily the identity. The interpretation $[\![M]\!]$ of each program M is just an element of 2.

▶ **Example 12.** For divergence, we use $\mathbf{C} = \omega\mathbf{Cpo}$. The strong $\mathbf{Poset}$-monad T freely adjoins a least element $\bot$ to each $\omega\mathbf{Cpo}$. The unit η_X is the inclusion $X \hookrightarrow TX$, while Kleisli extension is given by

$$f^{\dagger}(w, x) = \begin{cases} \bot & \text{if } x = \bot \\ f(w, x) & \text{otherwise} \end{cases}$$

A T-algebra Z is equivalently an $\omega\mathbf{Cpo}$ Z with a least element $\bot \in Z$. The extension operator is completely determined once the carrier is fixed; it is analogous to $(-)^{\dagger}$. Since the exponential $Y \Rightarrow Z$ is the set of ω-continuous functions $Y \to Z$ ordered pointwise, it has a least element (forms a T-algebra) whenever Z does.

If Z is a T-algebra, then every ω-continuous function $f : Z \to Z$ has a least fixed point $\mathit{fix}\, f = \bigsqcup_{n \in \mathbb{N}} f^n \bot \in Z$. These enable us to interpret recursive computations, by defining $[\![\mathbf{rec}\, x{:}\mathbf{U}\underline{C}.\, M]\!]\rho = \mathit{fix}\big(x \mapsto [\![M]\!](\rho, x)\big)$. The interpretation $[\![M]\!]$ of a program $M : \mathbf{F}\,\mathbf{bool}$ is either $\bot$ (signifying divergence), or one of the two elements of 2.

▶ **Example 13.** For finite nondeterminism, we use $\mathbf{C} = \mathbf{Poset}$. The strong $\mathbf{Poset}$-monad T freely adds finite joins to each poset. It is defined by

$$TX = (\{\downarrow S' \mid S' \in \mathcal{P}_{\mathrm{fin}}X\}, \subseteq) \qquad \eta_X\, x = \downarrow\{x\} \qquad f^{\dagger}(w, S) = \textstyle\bigcup_{x \in S} f(w, x)$$

where $\mathcal{P}_{\mathrm{fin}}X$ is the set of finite subsets of X, and $\downarrow S' = \{x \in X \mid \exists x' \in S'.\, x \sqsubseteq x'\}$ is the *downwards-closure* of $S' \subseteq X$. Each T-algebra is again completely determined by its carrier; a T-algebra Z is equivalently a poset Z that has finite joins. The extension operator is necessarily given by $f^{\ddagger}(w, S) = \bigsqcup_{x \in S} f(w, x)$. (The latter join exists because S is the downwards-closure of a finite set, even though S itself might not be finite.) The function space $Y \Rightarrow Z$ is the set of monotone functions $Y \to Z$, ordered pointwise; if Z has finite joins then $Y \Rightarrow Z$ has finite joins computed pointwise.

We interpret nondeterministic computations using nullary and binary joins:

$$[\![\mathbf{fail}_{\underline{C}}]\!]\rho = \bot \qquad [\![M \ \mathbf{or}\ N]\!]\rho = [\![M]\!]\rho \sqcup [\![N]\!]\rho$$

The interpretation $[\![M]\!]$ of a program $M : \mathbf{F}\,\mathbf{bool}$ is one of the four subsets of 2.

4 A Galois connection between call-by-value and call-by-name

We now return to the main contribution of this paper: relating call-by-value and call-by-name. The main difficulty in doing this is that the call-by-value and call-by-name translations of expressions have different types:

$$(\!|\Gamma|\!)^{\mathrm{v}} \vdash_c (\!|e|\!)^{\mathrm{v}} : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \qquad (\!|\Gamma|\!)^{\mathrm{n}} \vdash_c (\!|e|\!)^{\mathrm{n}} : (\!|\tau|\!)^{\mathrm{n}}$$

We cannot ask whether $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ are related by $\preccurlyeq_{\mathrm{ctx}}$, because $\preccurlyeq_{\mathrm{ctx}}$ only relates terms of the same type. It does not make sense to replace $(\!|e|\!)^{\mathrm{v}}$ with $(\!|e|\!)^{\mathrm{n}}$ inside a CBPV program, because the result would not be well-typed.

Reynolds [25] solves a similar problem when comparing direct and continuation semantics of the λ-calculus by defining maps between the two semantics, so that a denotation in the direct semantics can be viewed as a denotation in the continuation semantics and vice versa. We use the same idea here. Specifically, we define maps Φ from call-by-value computations to call-by-name computations, and Ψ from call-by-name to call-by-value:[2]

$$\Gamma \vdash_c M : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \ \mapsto\ \Gamma \vdash_c \Phi_\tau M : (\!|\tau|\!)^{\mathrm{n}} \qquad\qquad \Gamma \vdash_c N : (\!|\tau|\!)^{\mathrm{n}} \ \mapsto\ \Gamma \vdash_c \Psi_\tau N : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

Then instead of replacing $(\!|e|\!)^{\mathrm{v}}$ with $(\!|e|\!)^{\mathrm{n}}$ directly, we use Φ and Ψ to convert $(\!|e|\!)^{\mathrm{n}}$ to a computation of the same type as $(\!|e|\!)^{\mathrm{v}}$ (we define this computation formally in Section 5):

$$(\!|\Gamma|\!)^{\mathrm{v}} \longrightarrow (\!|\Gamma|\!)^{\mathrm{n}} \xrightarrow{(\!|e|\!)^{\mathrm{n}}} (\!|\tau|\!)^{\mathrm{n}} \longrightarrow \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

This behaves like a call-by-name computation, but has the same type as a call-by-value computation. We do not want just *any* maps between call-by-value and call-by-name. We show that, under certain conditions (the assumptions of Theorem 17 below) the maps we define form *Galois connections* [18] in the denotational semantics. This is crucial for the correctness of our reasoning principle. The conditions needed to show that we have Galois connections are where the choice of side-effects becomes important.

▶ **Definition 14.** *A* Galois connection *consists of two posets X, Y and two monotone functions $\phi : X \to Y$, $\psi : Y \to X$, such that $x \sqsubseteq \psi(\phi\, x)$ for all $x \in X$ and $\phi(\psi\, y) \sqsubseteq y$ for all $y \in Y$.*

[2] We define Φ and Ψ directly as maps from computations to computations, but we could instead have defined computations

$$x : \mathbf{U}\,\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}} \vdash_c \Phi'_\tau : (\!|\tau|\!)^{\mathrm{n}} \qquad x : \mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}} \vdash_c \Psi'_\tau : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

and then recovered Φ and Ψ using substitution (up to the equational theory defined in Appendix A). This definition is slightly less convenient to work with however.

The syntactic maps Φ_τ and Ψ_τ are defined as follows. (We use some extra variables in the definition, which are assumed to be fresh.)

$$\Phi_\tau M = M \textbf{ to } x.\,\hat{\Phi}_\tau x$$

$$\hat{\Phi}_{\mathbf{bool}} V = \mathbf{return}\, V \qquad \hat{\Phi}_{\tau\to\tau'} V = \lambda x{:}\mathbf{U}\,(\!|\tau|\!)^{\mathrm{n}}.\,\Psi_\tau(\mathbf{force}\, x) \textbf{ to } y.\ (y\,`\,\mathbf{force}\, V) \textbf{ to } z.\ \hat{\Phi}_{\tau'} z$$

$$\Psi_{\mathbf{bool}} N = N \qquad \Psi_{\tau\to\tau'} N = \mathbf{return}\,\mathbf{thunk}\,\lambda x{:}(\!|\tau|\!)^{\mathrm{v}}.\,\Psi_{\tau'}\big((\mathbf{thunk}\,(\hat{\Phi}_\tau x))\,`\,N\big)$$

The maps Φ_τ from call-by-value computations first evaluate the computation, and then map the result to call-by-name using $\hat{\Phi}_\tau$, which has the following typing:

$$\Gamma \vdash V : (\!|\tau|\!)^{\mathrm{v}} \quad \mapsto \quad \Gamma \vdash_c \hat{\Phi}_\tau V : (\!|\tau|\!)^{\mathrm{n}}$$

Given any model of CBPV, we correspondingly define morphisms $\phi_\tau : T[\![\tau]\!]^{\mathrm{v}} \to U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}$ and $\psi_\tau : U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}} \to T[\![\tau]\!]^{\mathrm{v}}$ as follows (where $\hat{\phi}_\tau : [\![\tau]\!]^{\mathrm{v}} \to U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}$).

$$\phi_\tau = \hat{\phi}_\tau^{\ddagger} \qquad \hat{\phi}_{\mathbf{bool}} = \eta_2 \qquad \hat{\phi}_{\tau\to\tau'} = \Lambda((\phi_{\tau'} \circ ev)^{\ddagger} \circ (id_{[\![\tau\to\tau']\!]^{\mathrm{v}}} \times \psi_\tau))$$

$$\psi_{\mathbf{bool}} = id_{T2} \qquad \psi_{\tau\to\tau'} = \eta_{[\![\tau\to\tau']\!]^{\mathrm{v}}} \circ (\hat{\phi}_\tau \Rightarrow \psi_{\tau'})$$

These morphisms are the interpretations of Φ and Ψ in the following sense.

▶ **Lemma 15.** *Given any model of CBPV, if* $\Gamma \vdash_c M : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$ *then* $[\![\Phi_\tau M]\!] = \phi_\tau \circ [\![M]\!]$*, and if* $\Gamma \vdash_c N : (\!|\tau|\!)^{\mathrm{n}}$ *then* $[\![\Psi_\tau N]\!] = \psi_\tau \circ [\![N]\!]$.

Proof sketch. By induction on the type τ. Each case is an easy calculation. ◀

For the rest of this section, we show that in every model of CBPV that satisfies certain conditions (the assumptions of Theorem 17 below), the functions

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \quad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

form a Galois connection for every τ and $W \in |\mathbf{C}|$. This is the key step in the proof of our reasoning principle (Theorem 21).

First we note that we cannot expect these maps to form Galois connections in general. Consider what happens when we convert a CBPV computation $M : \mathbf{F}\,(\!|\mathbf{bool} \to \mathbf{bool}|\!)^{\mathrm{v}} = \mathbf{F}\,(\mathbf{U}\,(\mathbf{bool} \to \mathbf{F}\,\mathbf{bool}))$ to call-by-name and then back to call-by-value. The computation $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M)$ that results is essentially[3] the same as

$$\mathbf{return}\,\mathbf{thunk}\,\lambda x{:}\mathbf{bool}.\,M \textbf{ to } z.\,x\,`\,\mathbf{force}\,z$$

The computation M may cause side-effects before producing a (thunk of a) function; but $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M)$ does not. Thus in general (e.g. if side-effects include mutable state), we cannot expect to have $[\![M]\!] \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}} \circ \phi_{\mathbf{bool}\to\mathbf{bool}} \circ [\![M]\!]$, and hence cannot expect to have a Galois connection. The round-trip from call-by-value to call-by-name and back thunks the side-effects of M.

The inequality $[\![M]\!] \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}}(\phi_{\mathbf{bool}\to\mathbf{bool}}[\![M]\!])$ does however hold when $[\![M]\!]$ is *lax thunkable* in the following sense.

[3] Precisely, they are the same in the sense that $\Psi_{\mathbf{bool}\to\mathbf{bool}}(\Phi_{\mathbf{bool}\to\mathbf{bool}} M) \equiv \mathbf{return}\,\mathbf{thunk}\,\lambda x : \mathbf{bool}.\,M \textbf{ to } z.\,x\,`\,\mathbf{force}\,z$, where $\equiv$ is the equational theory defined in Appendix A.

▶ **Definition 16.** *Let* T *be a strong* **Poset***-monad on a cartesian* $\mathbf{C}$*. We say that a morphism* $f : X \to TY$ *is* lax thunkable *when* $T\eta_Y \circ f \sqsubseteq \eta_{TY} \circ f$*. We say that* T *is* lax idempotent[4] *when* $T\eta_Y \sqsubseteq \eta_{TY}$ *for all* $Y \in |\mathbf{C}|$ *(equivalently, when every such morphism is lax thunkable).*

Our notion of *lax thunkable* morphism is an inequational version of Führmann's [5] notion of *thunkable* morphism. We do not need it below, but we can give a corresponding definition for CBPV: a computation $\Gamma \vdash_c M : \mathbf{F}\,A$ is *lax thunkable* (with respect to a given $\preccurlyeq$) when

$$M\ \mathbf{to}\ x.\,\mathbf{return}\,\mathbf{thunk}\,\mathbf{return}\,x \ \preccurlyeq^{\Gamma}_{\mathrm{ctx}}\ \mathbf{return}\,\mathbf{thunk}\,M$$

This is the case in particular when the interpretation $[\![M]\!] : [\![\Gamma]\!] \to T[\![A]\!]$ of M, in an adequate model, is a lax thunkable morphism.

Assuming that T is lax idempotent is enough to achieve the goal of this section:

▶ **Theorem 17.** *Given a CBPV model in which* T *is lax idempotent, the functions*

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \qquad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

form a Galois connection for every source-language type τ *and object* $X \in |\mathbf{C}|$*.*

Proof sketch. By induction on the type τ. This is trivial for **bool**. For function types both of the required inequalities use the fact that T is lax idempotent. ◀

▶ **Example 18.** Each of our three example models is adequate, and has a lax idempotent T. For no side-effects, we use the identity monad, which is trivially lax idempotent because $T\eta_Y = id_Y = \eta_{TY}$. For divergence, the monad is lax idempotent because the left hand side of $T\eta_Y t \sqsubseteq \eta_{TY} t$ is $\bot$ when $t = \bot$ (intuitively, we can thunk diverging computations), and otherwise the two sides are equal. For nondeterminism, we have

$$T\eta_Y S = {\downarrow}\{{\downarrow}\{y\} \mid y \in S\} \subseteq {\downarrow}\{S\} = \eta_{TY} S$$

because ${\downarrow}\{y\} \subseteq S$ for every $y \in S$ (intuitively, we can postpone nondeterministic choices). Thus we obtain Galois connections in each of these three cases.

▶ Remark 19. Given an adequate model in which T is lax idempotent, it follows from Theorem 17 that the maps Φ_τ and Ψ_τ on terms form a Galois connection (with respect to $\preccurlyeq_{\mathrm{ctx}}$), by Lemma 15. In particular, we have

$$M \preccurlyeq_{\mathrm{ctx}} \Psi_\tau(\Phi_\tau M) \qquad \Phi_\tau(\Psi_\tau N) \preccurlyeq_{\mathrm{ctx}} N$$

Both of these inequalities are in general proper (they are not contextual equivalences). To see this, consider our divergence example, for which the above inequalities hold. For each $\underline{C}$, let $\Omega_{\underline{C}}$ be the diverging computation $\mathbf{rec}\,x : \mathbf{U}\underline{C}.\,\mathbf{force}\,x$ (which has type $\underline{C}$). Then if $\tau = \mathbf{bool} \to \mathbf{bool}$ and $M = \Omega_{\mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}}$, we do not have $M \succcurlyeq_{\mathrm{ctx}} \Psi_\tau(\Phi_\tau M)$, because for $\mathcal{E} = (\square\ \mathbf{to}\ f.\,\mathbf{return}\,\mathbf{false})$ the computation $\mathcal{E}[M]$ diverges but $\mathcal{E}[\Psi_\tau(\Phi_\tau M)] \Downarrow \mathbf{return}\,\mathbf{false}$. In this case we have $\Psi_\tau(\Phi_\tau M) \cong_{\mathrm{ctx}} \mathbf{return}\,\mathbf{thunk}\,\lambda x : \mathbf{bool}.\,\Omega_{\mathbf{F}\,\mathbf{bool}}$. For a counterexample to $\Phi_\tau(\Psi_\tau N) \succcurlyeq_{\mathrm{ctx}} N$, let $\tau = \mathbf{bool} \to \mathbf{bool}$ and $N = \lambda x{:}\mathbf{U}\mathbf{F}\,\mathbf{bool}.\,\mathbf{return}\,\mathbf{true}$. Then for $\mathcal{E}' = ((\mathbf{thunk}\,\Omega_{\mathbf{F}\,\mathbf{bool}})\,`\,\square)$, the computation $\mathcal{E}'[\Phi_\tau(\Psi_\tau N)]$ diverges but $\mathcal{E}'[N] \Downarrow \mathbf{return}\,\mathbf{true}$. Here we have $\Phi_\tau(\Psi_\tau N) \cong_{\mathrm{ctx}} \lambda x{:}\mathbf{U}\mathbf{F}\,\mathbf{bool}.\,\mathbf{force}\,x\ \mathbf{to}\ y.\,\mathbf{return}\,\mathbf{true}$.

[4] Lax idempotent **Poset**-monads are a special case of lax idempotent 2-monads, which are well-known, and are often called *Kock-Zöberlein* monads [8].

5 The reasoning principle

We now use the Galois connections defined in the previous section to relate the call-by-value and call-by-name translations of expressions, and arrive at our main reasoning principle.

Recall that we compose the call-by-name translation of each e with the maps Φ and Ψ defined above, to form a CBPV computation of the same type as the call-by-value translation:

$$(\!|\Gamma|\!)^{\mathrm{v}} \longrightarrow (\!|\Gamma|\!)^{\mathrm{n}} \xrightarrow{(\!|e|\!)^{\mathrm{n}}} (\!|\tau|\!)^{\mathrm{n}} \longrightarrow \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$$

We first define this composition precisely. The arrow on the right is just given by applying Ψ_τ. The arrow on the left is a substitution that maps terms in call-by-name contexts $(\!|\Gamma|\!)^{\mathrm{n}}$ to terms in call-by-value contexts $(\!|\Gamma|\!)^{\mathrm{v}}$. Given any source-language typing context $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$, we define $\hat{\Phi}_\Gamma \;=\; x_1 \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau_1} x_1), \ldots, x_n \mapsto \mathbf{thunk}\,(\hat{\Phi}_{\tau_n} x_n)$. If $\Gamma \vdash e : \tau$ then $(\!|\Gamma|\!)^{\mathrm{v}} \vdash_c \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big) : \mathbf{F}\,(\!|\tau|\!)^{\mathrm{v}}$, which has the same typing as $(\!|e|\!)^{\mathrm{v}}$. The statement we wish to prove is $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$. Again we reason using the denotational semantics. Given a CBPV model, the interpretation of the substitution $\hat{\Phi}_\Gamma$ is a morphism $\hat{\phi}_\Gamma : [\![\Gamma]\!]^{\mathrm{v}} \to [\![\Gamma]\!]^{\mathrm{n}}$, given by $\hat{\phi}_\diamond = id_1$ and $\hat{\phi}_{\Gamma,x:\tau} = \hat{\phi}_\Gamma \times \hat{\phi}_\tau$. If the model is adequate, to show our goal $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$, it suffices to show that $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$.

We show that this is the case directly using the properties of Galois connections, which allow us to push composition with ψ_τ into the structure of terms.

▶ **Lemma 20.** *In every CBPV model for which the functions*

$$\phi_\tau \circ - : \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}}) \to \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \quad \psi_\tau \circ - : \mathbf{C}(W, U_{\mathsf{T}}[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(W, T[\![\tau]\!]^{\mathrm{v}})$$

form Galois connections for all τ, W, we have $[\![e]\!]^{\mathrm{v}} \;\sqsubseteq\; \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ for all $\Gamma \vdash e : \tau$.

Proof sketch. By induction on the derivation of $\Gamma \vdash e : \tau$. We give just the case for function applications $e\,e'$; which shows where having Galois connections is useful. The two inequalities below both use properties of Galois connections. The equalities follow from properties of products, exponentials, and T-algebras.

$$\begin{aligned}
[\![e\,e']\!]^{\mathrm{v}} &= \big(ev^\dagger \circ \langle \pi_2, [\![e']\!]^{\mathrm{v}} \circ \pi_1\rangle\big)^\dagger \circ \langle id, [\![e]\!]^{\mathrm{v}}\rangle \\
&\sqsubseteq \psi_{\tau'} \circ \phi_{\tau'} \circ \big(ev^\dagger \circ \langle \pi_2, \psi_\tau \circ [\![e']\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma \circ \pi_1\rangle\big)^\dagger \circ \langle id, \psi_{\tau\to\tau'} \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma\rangle \\
&= \psi_{\tau'} \circ \Lambda^{-1}(\phi_{\tau\to\tau'} \circ \psi_{\tau\to\tau'} \circ [\![e]\!]^{\mathrm{n}}) \circ \langle id, [\![e']\!]^{\mathrm{n}}\rangle \circ \hat{\phi}_\Gamma \\
&\sqsubseteq \psi_{\tau'} \circ \Lambda^{-1}[\![e]\!]^{\mathrm{n}} \circ \langle id, [\![e']\!]^{\mathrm{n}}\rangle \circ \hat{\phi}_\Gamma \\
&= \psi_{\tau'} \circ [\![e\,e']\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma
\end{aligned}$$ ◀

Theorem 17 provides a sufficient condition for the maps between the two evaluation orders to form Galois connections. By combining this sufficient condition with the above lemma, we arrive at our reasoning principle, which we state formally as Theorem 21. Recall that a program relation $\preccurlyeq$ is a family of relations on CBPV programs, and that each program relation induces a contextual preorder $\preccurlyeq_{\mathrm{ctx}}$. Given any program relation $\preccurlyeq$, to show that the call-by-value and call-by-name translations of source-language expressions are related by $\preccurlyeq_{\mathrm{ctx}}$ it is enough to find an adequate model involving a lax idempotent T.

▶ **Theorem 21** (Relationship between call-by-value and call-by-name)**.** *Suppose we are given a program relation $\preccurlyeq$, and a model of CBPV that is adequate with respect to $\preccurlyeq$, and has a lax idempotent T. If $\Gamma \vdash e : \tau$ then*

$$(\!|e|\!)^{\mathrm{v}} \;\preccurlyeq_{\mathrm{ctx}}\; \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$$

Proof. Theorem 17 implies that ϕ and ψ form Galois connections, and then Lemma 20 implies $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$. The result follows from adequacy of the model and Lemma 15. ◀

The generality of this theorem comes from two sources. First, we consider arbitrary program relations $\preccurlyeq$. The only requirement on these is the existence of some adequate model in which morphisms are lax thunkable. Second, this theorem applies to terms that are open and have higher types, using the maps between the two evaluation orders. We obtain a corollary about source-language *programs* (closed expressions of type **bool**). This corollary is closer to the standard results that are proved for specific side-effects.

▶ **Corollary 22.** *If the assumptions of Theorem 21 hold, then for every closed expression e of type* **bool**, *we have $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$.*

Proof. We have $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_{\mathbf{bool}} \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\diamond = [\![e]\!]^{\mathrm{n}}$ because both $\psi_{\mathbf{bool}}$ and $\hat{\phi}_\diamond$ are identities. Adequacy implies $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} (\!|e|\!)^{\mathrm{n}}$, and hence $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$. ◀

Our reasoning principle also has a partial converse:

▶ **Lemma 23.** *If $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ for each $\Gamma \vdash e : \tau$, then $T\eta_2 \sqsubseteq \eta_{T2}$, and every morphism $X \to T2$ is lax thunkable.*

Proof sketch. The first step is to show that $id \sqsubseteq \psi_\tau \circ \phi_\tau$ for every τ, by applying the assumption to the expression $x : \mathbf{bool} \to \tau \vdash x\,\mathbf{false} : \tau$. (It does not matter whether we use **false** or use **true**; we could have used $x : \mathbf{unit} \to \tau$ if we had a unit type.) Then, since $\phi_{\mathbf{bool}}$ and $\psi_{\mathbf{bool}}$ are identities, we get $id_{T(2\Rightarrow T2)} \sqsubseteq \psi_{\mathbf{bool}\to\mathbf{bool}} \circ \phi_{\mathbf{bool}\to\mathbf{bool}} = \eta_{2\Rightarrow T2} \circ id_{2\Rightarrow T2}{}^{\ddagger}$, from which the result follows. ◀

As a final remark, while we compose the call-by-value translation on both sides, this choice is in fact arbitrary. By properties of Galois connections, the inequality $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$ is equivalent to $\phi_\tau \circ [\![e]\!]^{\mathrm{v}} \sqsubseteq [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$, and two other inequalities are available when T is lax idempotent by defining suitable morphisms $\psi_\Gamma : [\![\Gamma]\!]^{\mathrm{n}} \to T[\![\Gamma]\!]^{\mathrm{v}}$.

We now return to our three examples. For each example, we take the adequate model defined in Section 3; in all three cases, the strong **Poset**-monad T is lax idempotent. After extending the inductive proof of Lemma 20 with cases for the extra syntax, we can apply our relationship between call-by-value and call-by-name (Theorem 21).

In particular, we can apply Corollary 22 to relate source-language programs. For no side-effects, this shows for each $e : \mathbf{bool}$ that there is some V such that $(\!|e|\!)^{\mathrm{v}} \Downarrow \mathbf{return}\,V$ and $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,V$. In other words, e evaluates to the same result in call-by-value and in call-by-name (since evaluation is deterministic). For divergence and for nondeterminism, the corollary says that $(\!|e|\!)^{\mathrm{v}} \Downarrow \mathbf{return}\,V$ implies $(\!|e|\!)^{\mathrm{n}} \Downarrow \mathbf{return}\,V$ for all V. Hence for divergence, if the call-by-value execution terminates with some result, the call-by-name execution terminates with the same result. For nondeterminism, all possible results of call-by-value executions are possible results of call-by-name executions.

6 Related work

Comparing evaluation orders Plotkin [24] and many others (e.g. [7]) relate call-by-value and call-by-name. Crucially, they consider lambda-calculi with no side-effects other than divergence. This makes a significant difference to the techniques that can be used, in particular because in this case the equational theory for call-by-name is strictly weaker than for call-by-value. This is not necessarily true for other side-effects. Other evaluation orders (such as call-by-need) have also been compared in similarly restricted settings [14, 15, 6]. We suspect our technique could also be adapted to these. Here we use CBPV as a calculus in which to reason about both call-by-value and call-by-name, but other calculi (e.g. the modal calculus of [28]) may be suitable for this purpose.

It might also be possible to recast some of our work in terms of the *duality* between call-by-value and call-by-name [3, 2, 30, 29], In particular, this may shed some light on our definitions of Φ and Ψ. It is not clear to us what the precise connection is however. While Selinger [29] defines translations between call-by-value and call-by-name versions of Parigot's $\lambda\mu$-calculus [23], these translations behave differently to ours, in particular, they are semantics-preserving.

Relating semantics of languages The technique we use here to relate call-by-value and call-by-name is based on the idea used first by Reynolds [25] to relate direct and continuation semantics of the lambda calculus, and later used by others (e.g. [19, 9, 1, 4]). There are several differences with our approach. Reynolds constructs a logical relation between the two semantics, and uses this to establish a relationship with the two maps. We skip the logical relation step. Reynolds also relies on continuations with a large-enough domain of answers (e.g. a solution to a particular recursive domain equation). Our maps exist for *any* choice of model. We are the first to use this technique to relate call-by-value and call-by-name. There has been some work [26, 10, 27] on soundness and completeness properties of translations (similar to the translations into CBPV), in particular using Galois connections (and similar structures) for which the order is reduction of programs. Our results would fail if we used reduction of programs directly, so we consider only the observable behaviour of programs.

There are some similarities between our work and the work of New et al. [21, 22] on gradual typing. In particular, [22] has embedding-projection pairs (a special case of Galois connections) for casting from a more dynamic type to a less dynamic type, and vice versa. Their application is quite different however. The double category perspective used in [21] may also be illuminating here.

7 Conclusions

In this paper, we give a general reasoning principle (Theorem 21) that relates the observable behaviour of terms under call-by-value and call-by-name. The reasoning principle works for various collections of side-effects, in particular, it enables us to obtain theorems about divergence and nondeterminism. It is about open expressions, and allows us to change evaluation order *within* programs. We obtain a result about call-by-value and call-by-name evaluations of *programs* as a corollary (Corollary 22). Applying this to divergence, we show that if the call-by-value execution terminates with some result then the call-by-name execution terminates with the same result. For nondeterminism, we show that all possible results of call-by-value executions are possible results of call-by-name executions. There may be other collections of side-effects we can apply our technique to, including combinations of divergence and nondeterminism.

We expect that our technique can be applied to other evaluation orders. Two evaluation orders can be related by giving translations into some common language (here we use CBPV), constructing maps between the two translations, and showing that (for some models) these maps form Galois connections. A major advantage of the technique is that it allows us to identify axiomatic properties of side-effects (thunkable, etc.) that give rise to relationships between evaluation orders.

References

1. Robert Cartwright and Matthias Felleisen. Extensible denotational language specifications. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 244–272. Springer, 1994. doi:10.1007/3-540-57887-0_99.
2. Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 233–243. ACM, 2000. doi:10.1145/351240.351262.
3. Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, 1989.
4. Andrzej Filinski. *Controlling Effects*. PhD thesis, Carnegie Mellon University, 1996.
5. Carsten Führmann. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science*, 20:245–292, 1999. doi:10.1016/S1571-0661(04)80078-1.
6. Jennifer Hackett and Graham Hutton. Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.*, 3(ICFP):114:1–114:23, 2019. doi:10.1145/3341718.
7. Jun Inoue and Walid Taha. Reasoning about multi-stage programs. *Journal of Functional Programming*, 26(e22), 2016. doi:10.1017/S0956796816000253.
8. Anders Kock. Monads for which structures are adjoint to units. *Journal of Pure and Applied Algebra*, 104(1):41–59, 1995. doi:10.1016/0022-4049(94)00111-U.
9. Jakov Kučan. Retraction approach to cps transform. *Higher Order Symbol. Comput.*, 11(2):145–175, 1998. doi:10.1023/A:1010012532463.
10. Julia L. Lawall and Olivier Danvy. Separating stages in the continuation-passing style transformation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 124–136. ACM, 1993. doi:10.1145/158511.158613.
11. Paul Blain Levy. Call-by-push-value: A subsuming paradigm. In Jean-Yves Girard, editor, *Typed Lambda Calculi and Applications*, pages 228–243. Springer, 1999. doi:10.1007/3-540-48959-2_17.
12. Paul Blain Levy. Adjunction models for call-by-push-value with stacks. *Electronic Notes in Theoretical Computer Science*, 69:248–271, 2003. CTCS'02, Category Theory and Computer Science. doi:https://doi.org/10.1016/S1571-0661(04)80568-1.
13. Paul Blain Levy. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation*, 19(4):377–414, 2006. doi:10.1007/s10990-006-0480-6.
14. John Maraist, Martin Odersky, David N. Turner, and Philip Wadler. 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*, pages 370–392, 1995. doi:10.1016/S1571-0661(04)00022-2.
15. Dylan McDermott and Alan Mycroft. Extended call-by-push-value: Reasoning about effectful programs and evaluation order. In Luís Caires, editor, *Programming Languages and Systems*, pages 235–262. Springer, 2019. doi:10.1007/978-3-030-17184-1_9.
16. Dylan McDermott and Alan Mycroft. Galois connecting call-by-value and call-by-name (extended version), 2022. arXiv:2202.08246.
17. Dylan McDermott and Tarmo Uustalu. What makes a strong monad? In *Proceedings Ninth Workshop on Mathematically Structured Functional Programming (to appear)*. Open Publishing Association, 2022.
18. Austin Melton, David A. Schmidt, and George E. Strecker. Galois connections and computer science applications. In *Category Theory and Computer Programming*, pages 299–312. Springer, 1986. doi:10.1007/3-540-17162-2_130.
19. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi. In Rohit Parikh, editor, *Logics of Programs*, pages 219–224. Springer, 1985. doi:10.1007/3-540-15648-8_17.
20. Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi:10.1016/0890-5401(91)90052-4.

21 Max S. New and Daniel R. Licata. Call-by-name gradual type theory. *Logical Methods in Computer Science*, 16, 2020. doi:10.23638/LMCS-16(1:7)2020.

22 Max S. New, Daniel R. Licata, and Amal Ahmed. Gradual type theory. *Journal of Functional Programming*, 31, 2021. doi:10.1017/S0956796821000125.

23 Michel Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Andrei Voronkov, editor, *Logic Programming and Automated Reasoning*, pages 190–201. Springer, 1992. doi:10.1007/BFb0013061.

24 G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975. doi:10.1016/0304-3975(75)90017-1.

25 John C. Reynolds. On the relation between direct and continuation semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*, pages 141–156. Springer, 1974. doi:10.1007/978-3-662-21545-6_10.

26 Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 288–298. ACM, 1992. doi:10.1145/141471.141563.

27 Amr Sabry and Philip Wadler. A reflection on call-by-value. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, pages 13–24. ACM, 1996. doi:10.1145/232627.232631.

28 José Espírito Santo, Luís Pinto, and Tarmo Uustalu. Plotkin's call-by-value λ-calculus as a modal calculus. *Journal of Logical and Algebraic Methods in Programming*, 2022. doi:10.1016/j.jlamp.2022.100775.

29 Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001. doi:10.1017/S096012950000311X.

30 Philip Wadler. Call-by-value is dual to call-by-name. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 189–201. ACM, 2003. doi:10.1145/944705.944723.

31 Glynn Winskel. *The Formal Semantics of Programming Languages: An Introduction.* MIT Press, 1993. doi:10.7551/mitpress/3054.001.0001.

$$\begin{aligned} \textbf{if true then } M_1 \textbf{ else } M_2 &\equiv M_1 & V`\lambda x{:}A.\,M &\equiv M[x \mapsto V] \\ \textbf{if false then } M_1 \textbf{ else } M_2 &\equiv M_2 & \textbf{force}\,\textbf{thunk}\,M &\equiv M \end{aligned}$$

$$\begin{aligned} V &\equiv \textbf{thunk}\,\textbf{force}\,V \\ M[x \mapsto V] &\equiv \textbf{if } V \textbf{ then } M[x \mapsto \textbf{true}] \textbf{ else } M[x \mapsto \textbf{false}] \\ M &\equiv \lambda x{:}A.\,x`M \end{aligned}$$

$$\begin{aligned} &\textbf{return}\,V \textbf{ to } x.\,M \equiv M[x \mapsto V] \\ &M \equiv M \textbf{ to } x.\,\textbf{return}\,x \\ &(M_1 \textbf{ to } x.\,M_2) \textbf{ to } y.\,M_3 \equiv M_1 \textbf{ to } x.\,(M_2 \textbf{ to } y.\,M_3) \\ &\lambda y{:}A.\,M \textbf{ to } x.\,N \equiv M \textbf{ to } x.\,\lambda y{:}A.\,N \end{aligned}$$

Figure 5 (Typed) equations between CBPV terms

A CBPV equational theory

We define an equational theory for CBPV. We write $\equiv$ for the smallest equivalence relation on terms of the same type that is closed under the axioms in Figure 5 and under the syntax of CBPV terms (for example, $M \equiv N$ implies $\textbf{thunk}\,M \equiv \textbf{thunk}\,N$ and $V \equiv W$ implies $\textbf{return}\,V \equiv \textbf{return}\,W$). (This is not exactly Levy's equational theory for CBPV, because we do not include *complex values*.)

All of the axioms should be read as subject to suitable typing constraints. The group of axioms at the top of Figure 5 contains the β-laws for all of the type formers except $\mathbf{F}$. The second group contains η-laws. The bottom group contains axioms governing the behaviour of sequencing of computations: there is a left-unit axiom, a right-unit axiom, an associativity axiom, and axioms for commuting sequencing with the introduction form for functions.