On the relation between call-by-value and call-by-name

Dylan McDermott
Reykjavik University
dylanm@ru.is

Joint work with Alan Mycroft (University of Cambridge)

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. 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.

1 Introduction

Suppose that we have a program that can be evaluated using either call-by-value or call-by-name. If the language has side-effects then these two evaluation orders will give rise to different observable behaviours. For specific side-effects, we can often say something about how we expect them to differ:

- If there are no side-effects at all (in particular, all programs are strongly normalizing), the choice of call-by-value or call-by-name does not affect the behaviour of the program.
- If there are diverging terms (for instance, via recursion), then the behaviours may differ: a program might diverge under call-by-value and return a result under call-by-name. But if the program terminates with some result under call-by-value, then it terminates with the same result under call-by-name.
- If nondeterminism is the only side-effect, every possible result of a call-by-value execution is a possible result of a call-by-name execution, but the reverse is not true.

These properties are intuitively obvious, and can be proved for specific examples by reasoning at the meta-level. We develop a *general* technique for proving these properties.

We use a language (Levy's *call-by-push-value* [2]) that captures both call-by-value and call-by-name. Levy describes how to translate open source-language 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 program contexts.

$$\frac{\Gamma \vdash M : \underline{C}}{\Gamma \vdash \mathbf{thunk}\, M : \mathbf{U}\underline{C}} \qquad \frac{\Gamma \vdash V : \mathbf{U}\underline{C}}{\Gamma \vdash \mathbf{force}\, V : \underline{C}} \qquad \frac{\Gamma \vdash V : A}{\Gamma \vdash \mathbf{return}\, V : \mathbf{F}A} \qquad \frac{\Gamma \vdash M : \mathbf{F}A \quad \Gamma, x : A \vdash N : \underline{C}}{\Gamma \vdash M\ \mathbf{to}\ x.\, N : \underline{C}}$$

Figure 1: Some of the CBPV typing rules

The main difficulty in doing this is that $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ have different types (and hence we cannot consider whether they are e.g. contextually equivalent). The call-by-value and call-by-name translations have two different interpretations of source-language types. Our solution is inspired by work relating direct and continuation interpretations of languages [6]: we construct maps between the call-by-value and call-by-name interpretations, and then compose these with the translations of expressions to arrive at two terms that can be compared directly. We show that, under certain conditions, the maps between call-by-value and call-by-name form a *Galois connection*. This gives rise to a general reasoning principle that allows us to compare call-by-value and call-by-name evaluation. We apply this reasoning principle to examples by choosing $\preccurlyeq$ to mirror the properties described informally above.

Rather than considering some fixed collection of side-effects, we work abstractly and identify properties of side-effects that enable us to relate call-by-value and call-by-name. We work with *adjunction models* for CBPV [3]; there are well-known models for various side-effects. Crucially, we use an *order-enriched* notion of adjunction model. In these models denotations are ordered; this is necessary for our examples because of the asymmetry in the properties above.

We believe it might be possible to use the technique we describe here be used to study the relationship between other evaluation orders by deriving similar reasoning principles. Relating call-by-value and call-by-name is just one example.

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

Levy [2, 4] introduced call-by-push-value (CBPV) as a language 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. Here we provide a brief overview of CBPV and the call-by-value and call-by-name translations (omitting many details).

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* $\underline{C}, \underline{D}$.

$$\begin{array}{rcl} A, B & ::= & \mathbf{U}\underline{C} \mid \ldots \\ \underline{C}, \underline{D} & ::= & A \to \underline{C} \mid \mathbf{F}A \mid \ldots \\ V, W & ::= & \mathbf{thunk}\, M \mid \ldots \\ M, N & ::= & \mathbf{force}\, V \mid \lambda x{:}A.\, M \mid V \text{`} M \mid \mathbf{return}\, V \mid M\ \mathbf{to}\ x.\, N \mid \ldots \end{array}$$

Value types include *thunk types* $\mathbf{U}\underline{C}$, which contain suspended computations of type $\underline{C}$. Computation types include function types (where functions send values to computations), and *returner types* $\mathbf{F}A$. The latter contains computations that return elements of the value type A; these computations may have side-effects. Thunks are introduced using **thunk** and eliminated using **force**. Function application is written $V \text{`} M$, where V is the argument and M is the function to apply. The introduction form for returner types is **return** V, which is a computation that

$$\begin{array}{rcl} \text{types } \tau & \mapsto & \text{value types } (\!|\tau|\!)^{\mathrm{v}} \\ \mathbf{bool} & \mapsto & \mathbf{bool} \\ \tau \to \tau' & \mapsto & \mathbf{U}((\!|\tau|\!)^{\mathrm{v}} \to \mathbf{F}(\!|\tau'|\!)^{\mathrm{v}}) \end{array} \qquad \begin{array}{rcl} \text{typing contexts } \Gamma & \mapsto & \text{typing contexts } (\!|\Gamma|\!)^{\mathrm{v}} \\ \diamond & \mapsto & \diamond \\ \Gamma, x : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{v}}, x : (\!|\tau|\!)^{\mathrm{v}} \end{array}$$

$$\text{expressions } \Gamma \vdash e : \tau \quad \mapsto \quad \text{computations } (\!|\Gamma|\!)^{\mathrm{v}} \sqsubseteq (\!|e|\!)^{\mathrm{v}} : \mathbf{F}(\!|\tau|\!)^{\mathrm{v}}$$

(a) Call-by-value translation $(\!|-|\!)^{\mathrm{v}}$

$$\begin{array}{rcl} \text{types } \tau & \mapsto & \text{computation types } (\!|\tau|\!)^{\mathrm{n}} \\ \mathbf{bool} & \mapsto & \mathbf{F}\,\mathbf{bool} \\ \tau \to \tau' & \mapsto & (\mathbf{U}(\!|\tau|\!)^{\mathrm{n}}) \to (\!|\tau'|\!)^{\mathrm{n}} \end{array} \qquad \begin{array}{rcl} \text{typing contexts } \Gamma & \mapsto & \text{typing contexts } (\!|\Gamma|\!)^{\mathrm{n}} \\ \diamond & \mapsto & \diamond \\ \Gamma, x : \tau & \mapsto & (\!|\Gamma|\!)^{\mathrm{n}}, x : \mathbf{U}(\!|\tau|\!)^{\mathrm{n}} \end{array}$$

$$\text{expressions } \Gamma \vdash e : \tau \quad \mapsto \quad \text{computations } (\!|\Gamma|\!)^{\mathrm{n}} \sqsubseteq (\!|e|\!)^{\mathrm{n}} : (\!|\tau|\!)^{\mathrm{n}}$$

(b) Call-by-name translation $(\!|-|\!)^{\mathrm{n}}$

Figure 2: Translations from the source language into the call-by-push-value calculus

immediately returns the value V (with no side-effects). The eliminator for returner types is the computation M **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.) Figure 1 gives some of the typing rules for these constructs.

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.

A CBPV *program* is a closed computation $M : \mathbf{F}G$, where G is a *ground type*, meaning a value type that does not contain thunks (e.g. **unit** or **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 $\preccurlyeq$ on closed computations of type $\mathbf{F}G$ for each ground type G. For example, we could use

$$M \preccurlyeq M' \quad \text{iff} \quad \forall V : G.\ (M \Downarrow \mathbf{return}\, V) \implies (M' \Downarrow \mathbf{return}\, V)$$

where $M \Downarrow T$ is big-step evaluation of computations. (We do not choose a specific $\preccurlyeq$ at this point to keep the discussion general.) Given any program relation $\preccurlyeq$, we define a *contextual preorder* $M \preccurlyeq_{\mathrm{ctx}} M'$ on arbitrary well-typed computations in the usual way: $M \preccurlyeq_{\mathrm{ctx}} M'$ holds if $C[M] \preccurlyeq C[M']$ for all suitably-typed contexts $C[\,]$.

We use CBPV because it captures both call-by-value and call-by-name evaluation. Levy [2] gives two compositional translations from a source language into CBPV: one for call-by-value and one for call-by-name. Our goal is to reason about the relationship between these two translations.

Figure 2 gives excerpts of the two translations from the source language to CBPV. For call-by-value, each source language type τ maps 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, hence the argument must be evaluated before passing it to the function. Under the call-by-name translation, functions accept thunks of computations as arguments. Arguments are not evaluated before passing them to call-by-name functions. Source-language expressions e are mapped to CBPV computations $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$.

2.1 Order-enriched models of CBPV

To capture non-symmetric program relations $\preccurlyeq$, we need to use models of CBPV that order denotations. Hence we work with a **Poset**-enriched version of Levy's *adjunction models* of CBPV [3].

We omit most of the details, but note that the core of the definition of adjunction model involves two **Poset**-categories (categories in which each hom-set is partially ordered by a specified relation $\sqsubseteq$ and composition is monotone). These are the *value category* $\mathbf{C}$, which contains the interpretation of each value type, and the *computation category* $\mathbf{D}$, which contains the interpretation of each computation type. Models also contain functors $F : \mathbf{C} \to \mathbf{D}$ (to interpret returner types $\mathbf{F}A$) and $U : \mathbf{D} \to \mathbf{C}$ (to interpret thunk types $\mathbf{U}\underline{C}$). As the name suggests, we require these to form an adjunction $F \dashv U$ (which allows us to interpret $\mathbf{return}\, V$ and M $\mathbf{to}$ $x.\, N$). We give three examples (one for each of the bullet points in the introduction).

Example 1 For a language with no side-effects, we can take $\mathbf{C} = \mathbf{D} = \mathbf{Set}$ (with $\sqsubseteq$ the discrete order) and $F = U = Id_{\mathbf{Set}}$. ◀

Example 2 To interpret fixed-points, we use the category of ωcpos and continuous functions for $\mathbf{C}$, and the category of pointed ωcpos and strict continuous functions for $\mathbf{D}$. In both cases morphisms are ordered pointwise. The functor F sends each ωcpo X to the free pointed ωcpo $X_\perp$ on X, and U is the forgetful functor. ◀

Example 3 For finitary nondeterminism, we use the category of posets and monotone functions for $\mathbf{C}$, and the category of bounded join-semilattices and join-preserving monotone functions for $\mathbf{D}$. Again morphisms are ordered pointwise, and the adjunction $F \dashv U$ is free/forgetful. ◀

Given an adjunction model, we interpret value types A as objects $[\![A]\!] \in \mathbf{C}$ and computation types as objects $[\![\underline{C}]\!] \in \mathbf{D}$, by defining in particular $[\![\mathbf{F}A]\!] = F[\![A]\!]$ and $[\![\mathbf{U}\underline{C}]\!] = U[\![\underline{C}]\!]$. Typing contexts Γ are interpreted as objects $[\![\Gamma]\!] \in \mathbf{C}$. Value terms $\Gamma \vdash V : A$ are then interpreted as morphisms $[\![V]\!] : [\![\Gamma]\!] \to [\![A]\!]$, and computation terms $\Gamma \vdash\!\!\!: M : \underline{C}$ as morphisms $[\![M]\!] : [\![\Gamma]\!] \to U[\![\underline{C}]\!]$ (these are both in $\mathbf{C}$).

The reasoning principle we derive proves instances of contextual preorders; for this we need a *adequate* model.

Definition 4 A model of CBPV is *adequate* (with respect to a program relation $\preccurlyeq$) if for all computations $\Gamma \vdash\!\!\!: M : \underline{C}$ and $\Gamma \vdash\!\!\!: M' : \underline{C}$ we have

$$[\![M]\!] \sqsubseteq [\![M']\!] \quad \Rightarrow \quad M \preccurlyeq_{\mathrm{ctx}} M'$$ ◀

Adequacy holds for the example models above.

$$
\begin{aligned}
\Phi_\tau M &\coloneqq M \,\mathbf{to}\, x.\, \hat{\Phi}_\tau x \\
\\
\hat{\Phi}_{\mathbf{bool}} V &\coloneqq \mathbf{return}\, V \\
\hat{\Phi}_{\tau\to\tau'} V &\coloneqq \lambda x{:}\mathbf{U}(\!|\tau|\!)^{\mathrm{n}}.\, \Psi_\tau(\mathbf{force}\, x)\ \mathbf{to}\ z.\ z\text{`}\mathbf{force}\, V\ \mathbf{to}\ z'.\ \hat{\Phi}_{\tau'} z' \\
\\
\Psi_{\mathbf{bool}} N &\coloneqq N \\
\Psi_{\tau\to\tau'} N &\coloneqq \mathbf{return}\,\mathbf{thunk}\,\lambda x{:}(\!|\tau|\!)^{\mathrm{v}}.\, \Psi_{\tau'}\big((\mathbf{thunk}\,(\hat{\Phi}_\tau x))\text{`}N\big)
\end{aligned}
$$

Figure 3: Maps Φ from call-by-value to call-by-name and Ψ from call-by-name to call-by-value

3 Galois connection between call-by-value and call-by-name

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

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

We cannot ask whether $(\!|e|\!)^{\mathrm{v}}$ and $(\!|e|\!)^{\mathrm{n}}$ are related by the contextual preorder, because it 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.

A similar problem arises when comparing two different denotational semantics of the same language. When comparing direct and continuation semantics of the lambda calculus, Reynolds [6], solves this problem 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 a similar 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:

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

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 correct type (defined formally in Section 4):

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

This behaves like a call-by-name computation, but has a call-by-value type. The maps Φ_τ and Ψ_τ are defined by induction on τ (example cases are given in Figure 3).

It is not enough to have just *any* maps between call-by-value and call-by-name. To derive our reasoning principle we use the fact that, in any model of CBPV that satisfies certain conditions, the interpretations of Φ_τ and Ψ_τ form a Galois connection [5] for each τ.

The interpretations of Φ_τ and Ψ_τ are morphisms $\phi_\tau : F[\![\tau]\!]^{\mathrm{v}} \to [\![\tau]\!]^{\mathrm{n}}$ in the computation category $\mathbf{D}$ and $\psi_\tau : U[\![\tau]\!]^{\mathrm{n}} \to U(F[\![\tau]\!]^{\mathrm{v}})$ in the value category $\mathbf{C}$. (We write $[\![-]\!]^{\mathrm{v}}$ for $[\![(\!|-|\!)^{\mathrm{v}}]\!]$, and similarly for call-by-name.) These satisfy $[\![\Phi_\tau M]\!] = U\phi_\tau \circ [\![M]\!]$ and $[\![\Psi_\tau N]\!] = \psi_\tau \circ [\![N]\!]$. Using composition, ϕ_τ and ψ_τ induce monotone functions between posets of morphisms:

$$U\phi_\tau \circ -\ :\ \mathbf{C}(X, U(F[\![\tau]\!]^{\mathrm{v}})) \to \mathbf{C}(X, U[\![\tau]\!]^{\mathrm{n}}) \qquad \psi_\tau \circ -\ :\ \mathbf{C}(X, U[\![\tau]\!]^{\mathrm{n}}) \to \mathbf{C}(X, U(F[\![\tau]\!]^{\mathrm{v}}))$$

Our goal is to show that these functions form a Galois connection for each τ, i.e. that

$$U\phi_\tau \circ \psi_\tau \sqsubseteq id_{U[\![\tau]\!]^{\mathrm{n}}} \qquad\qquad id_{UF[\![\tau]\!]^{\mathrm{v}}} \sqsubseteq \psi_\tau \circ U\phi_\tau$$

By looking at these inequalities in the syntax, we see that they do not hold in general. Consider what happens when a function is converted from call-by-value to call-by-name and back: the computation $\Psi_{\tau\to\tau'}(\Phi_{\tau\to\tau'}M)$ is pure (has no side-effects), even if M has side-effects. The round-trip between the two evaluation orders *thunks* the side-effects of M, so that they do not occur until the function is applied.

Based on this observation, we restrict our attention to models in which morphisms are *(lax) thunkable*. This property was first defined in a symmetric setting by Führmann [1]. It is where we need to be careful about the side-effects: morphisms are only thunkable in certain cases (they are for each our examples, but not e.g. for mutable state). We give three related definitions of thunkable: for computation terms, for morphisms, and for the model itself.

Definition 5 A computation $\Gamma \vdash M : \mathbf{F}A$ is *(lax) thunkable* if

$$M \text{ to } x.\ \mathbf{return}\ \mathbf{thunk}\ \mathbf{return}\ x \ \preccurlyeq_{\text{ctx}}\ \mathbf{return}\ \mathbf{thunk}\ M$$

A morphism $f : X \to UFY$ in $\mathbf{C}$ is *(lax) thunkable* if $UF\eta_Y \circ f \sqsubseteq \eta_{UFY} \circ f$ (where η is the unit of the adjunction $F \dashv U$). A model of CBPV is *(lax) thunkable* if $UF\eta_Z \sqsubseteq \eta_{UFZ}$ for all $Z \in \mathbf{C}$. ◂

The three definitions are related as follows: a model is thunkable if and only if all morphisms are thunkable, and for adequate models a computation M is thunkable if the morphism $[\![M]\!]$ is. Thunkable is a strong property (though still weaker than the symmetric version considered by Führmann). In particular, thunkable morphisms can be discarded (giving an instance of $\sqsubseteq$) if their results are not used, and duplicates can be merged. Moreover, any two thunkable computations can be commuted without changing the behaviour of a computation.

Our maps between call-by-value and call-by-name do indeed form Galois connections when the model is thunkable:

Theorem 6 Suppose a thunkable model of CBPV. For every source-language type τ we have:

$$U\phi_\tau \circ \psi_\tau \sqsubseteq id_{U[\![\tau]\!]^{\mathrm{n}}} \qquad\qquad id_{UF[\![\tau]\!]^{\mathrm{v}}} \sqsubseteq \psi_\tau \circ U\phi_\tau$$ ◂

(We prove this for source-language types including products, coproducts and function types.) Our example models are thunkable, and we therefore have Galois connections in those cases.

4 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, and arrive at our main reasoning principle.

Recall that we compose the call-by-name translation of each expression e with the maps Φ and Ψ, to arrive at 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 $\hat{\Phi}_\Gamma$ from terms in call-by-name contexts $(\!|\Gamma|\!)^{\mathrm{n}}$ to terms in call-by-value contexts $(\!|\Gamma|\!)^{\mathrm{v}}$. We define $\hat{\Phi}_\Gamma$ for $\Gamma = x_1 : \tau_1, \ldots x_n : \tau_n$ as

$$\hat{\Phi}_\Gamma \ :=\ x_1 \mapsto \mathbf{thunk}\left(\hat{\Phi}_{\tau_1} x_1\right), \ldots, x_n \mapsto \mathbf{thunk}\left(\hat{\Phi}_{\tau_n} x_n\right)$$

If $\Gamma \vdash e : \tau$ then $(\!|\Gamma|\!)^{\mathrm{v}} \vdash \Psi_\tau\left((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\right) : \mathbf{F}(\!|\tau|\!)^{\mathrm{v}}$. This term has the same typing as $(\!|e|\!)^{\mathrm{v}}$, and the statement we wish to prove is

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

Again we reason using the denotational semantics, so we say what this means inside the model. We interpret the substitution $\hat{\Phi}_\Gamma$ as a morphism $\hat{\phi}_\Gamma : [\![\Gamma]\!]^{\mathrm{v}} \to [\![\Gamma]\!]^{\mathrm{n}}$. The interpretations of the computations we wish to relate are the two sides of the inequality $[\![e]\!]^{\mathrm{v}} \sqsubseteq \psi_\tau \circ [\![e]\!]^{\mathrm{n}} \circ \hat{\phi}_\Gamma$, so if the model is adequate, this inequality implies the above instance of the contextual preorder. The inequality can be proved directly using the properties of Galois connections, which allow us to push composition with ψ_τ into the structure of terms.

We therefore arrive at our reasoning principle, which we state formally as Theorem 7. 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 define a model, and then show that this model is adequate and thunkable.

Theorem 7 (Relationship between call-by-value and call-by-name) Suppose given a program relation $\preccurlyeq$ and a model of CBPV that is both thunkable and adequate with respect to $\preccurlyeq$. If $\Gamma \vdash e : \tau$ then $(\!|e|\!)^{\mathrm{v}} \preccurlyeq_{\mathrm{ctx}} \Psi_\tau\big((\!|e|\!)^{\mathrm{n}}[\hat{\Phi}_\Gamma]\big)$. ◂

(We show this for a source-language including products, coproducts and functions, as well as fixed points and finitary nondeterminism.)

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 thunkable. Second, this theorem applies to terms that are open and have higher types, using the maps between the two evaluation orders. A corollary is that (if the conditions of the reasoning principle hold) then for source-language *programs* e, we have $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$.

We now return to our three examples (the bullet points in the introduction). The three models outlined in Section 2.1 are thunkable, and are adequate with respect to

$$M \preccurlyeq M' \quad \text{iff} \quad \forall V : G.\ (M \Downarrow \mathbf{return}\, V) \implies (M' \Downarrow \mathbf{return}\, V)$$

Hence our reasoning principle applies to each (for the second and third, when the source language includes fixed points and finitary nondeterminism respectively). Unpacking the corollary $(\!|e|\!)^{\mathrm{v}} \preccurlyeq (\!|e|\!)^{\mathrm{n}}$ for programs e gives us the three properties in the introduction.

5 Conclusions

We give a general reasoning principle (Theorem 7) that relates the 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 nontermination and nondeterminism. It is about open expressions, and allows changes of evaluation order *within* programs. We obtain a corollary about call-by-value and call-by-name evaluations of *programs*. Applying this to nontermination, 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.

We expect that our technique could 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.

Bibliography

[1] Carsten Führmann. 1999. Direct models of the computational lambda-calculus. *Electronic Notes in Theoretical Computer Science* 20 (1999), 245–292.

[2] Paul Blain Levy. 1999. Call-by-Push-Value: A Subsuming Paradigm. In *Typed Lambda Calculi and Applications*, Jean-Yves Girard (Ed.). Springer, Berlin, Heidelberg, 228–243. https://doi.org/10.1007/3-540-48959-2_17

[3] Paul Blain Levy. 2003. Adjunction Models For Call-By-Push-Value With Stacks. *Electronic Notes in Theoretical Computer Science* 69 (2003), 248–271. https://doi.org/10.1016/S1571-0661(04)80568-1 CTCS'02, Category Theory and Computer Science.

[4] Paul Blain Levy. 2006. Call-by-push-value: Decomposing call-by-value and call-by-name. *Higher-Order and Symbolic Computation* 19, 4 (01 Dec 2006), 377–414. https://doi.org/10.1007/s10990-006-0480-6

[5] A Melton, D A Schmidt, and G E Strecker. 1986. Galois Connections and Computer Science Applications. In *Proceedings of a Tutorial and Workshop on Category Theory and Computer Programming*. Springer-Verlag, Berlin, Heidelberg, 299–312. http://dl.acm.org/citation.cfm?id=20081.20099

[6] John C. Reynolds. 1974. On the Relation Between Direct and Continuation Semantics. In *Proceedings of the 2nd Colloquium on Automata, Languages and Programming*. Springer-Verlag, Berlin, Heidelberg, 141–156. http://dl.acm.org/citation.cfm?id=646230.681878