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

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 strongly 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).
Galois connecting call-by-value and call-by-name

The calculus we use is Levy’s call-by-push-value (CBPV) [13]. Levy describes how to translate (possibly open) expressions e into CBPV terms \( \langle e \rangle^v \) and \( \langle e \rangle^n \), which respectively correspond to call-by-value and call-by-name. We study the relationship between the behaviour of \( \langle e \rangle^v \) and the behaviour of \( \langle e \rangle^n \) in a given program context.

The main obstacle is that \( \langle e \rangle^v \) and \( \langle e \rangle^n \) have different types, and hence cannot be directly compared. Our solution to this is based on Reynold’s work relating direct and continuation semantics of the \( \lambda \)-calculus [24]: 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 22). This fact gives rise to a general reasoning principle (Theorem 25) that we use to compare call-by-value with call-by-name. Given any preorder \( \preceq \) that captures the property we wish to show about programs, our reasoning principle gives sufficiency conditions for showing \( M \preceq M' \), 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 \( \preceq \); each of these relations indicates the extent to which changing evaluation order affects the behaviour of the program. In the divergence example \( N \preceq N' \) is defined to mean termination of \( N \) implies termination of \( N' \) with the same result; in the other examples \( \preceq \) 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 5), and show that they form a Galois connection for side-effects satisfying certain conditions (Theorem 22).
- We use the Galois connection to prove a novel reasoning principle (Theorem 25) that relates the call-by-value and call-by-name translations of expressions (Section 6).

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 [13, 15] 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 $C,D$.

- **Value types**: $A,B ::= \text{unit} | A_1 \times A_2 | \text{bool} | UC$
- **Computation types**: $C,D ::= C_1 \times C_2 | A \to C | FA$
- **Values**: $V,W ::= x | () | (V_1,V_2) | \text{true} | \text{false} | \text{thunk } M$
- **Computations**: $M,N ::= \lambda\{1.M_1,2.M_2\} | 1'M | 2'M$
  - $| \lambda x : A.M | V'M | \text{return } V | M \text{ to } x.N$
  - $| \text{match } V \text{ with } (x_1,x_2). M$
  - $| \text{if } V \text{ then } M_1 \text{ else } M_2 | \text{force } V$

We restrict to only the subset of CBPV required for this paper.

The value type $UC$ is the type of *thunks* of computations of type $C$. Elements of $UC$ are introduced using thunk: the value thunk $M$ is the suspension of the computation term $M$. The corresponding eliminator is force, which is the inverse of thunk. Computation types include binary products; the pairing of two computations $M_1$ and $M_2$ is written $\lambda\{1.M_1,2.M_2\}$, and the first and second projections are $1'M$ and $2'M$. Computation types also 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 $FA$ has as elements computations that return elements of the value type $A$; these computations may have side-effects. Elements of $FA$ are introduced by return; the computation return $V$ immediately returns the value $V$ (with no side-effects). Computations can be sequenced using $M \text{ to } x.N$. 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 $M >>= \lambda x -> N$ 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 $\Gamma \vdash c : C$ for computations. Typing contexts $\Gamma$ 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 $M : C$ as an abbreviation for $\diamond \vdash c : M : C$.

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:

$$R ::= \lambda\{1.M_1,2.M_2\} | \lambda x : A.M | \text{return } V$$
Figure 1 CBPV typing rules
\[
\begin{align*}
\lambda\{1. M_1, 2. M_2\} & \Downarrow \lambda\{1. M_1, 2. M_2\} \\
\lambda x : A. M & \Downarrow \lambda x : A. M \\
\text{return } V & \Downarrow \text{return } V \\
\text{if true then } M_1 \text{ else } M_2 & \Downarrow R \\
\text{match } (V_1, V_2) \text{ with } (x_1, x_2). M & \Downarrow R \\
\text{force } (\text{thunk } M) & \Downarrow R
\end{align*}
\]

We only evaluate closed, well-typed computations, so when we write \( M \Downarrow R \) we assume \( M : \mathcal{C} \) for some \( \mathcal{C} \) (this implies \( R : \mathcal{C} \)). Reduction therefore cannot get stuck. The rules defining \( \Downarrow \) are given in Figure 2. All terminal computations evaluate to themselves. Products of computations are lazy: to evaluate a projection \( \pi_i M \), only the \( i \)th component of the pair \( M \) is evaluated. Since we have not yet included any way of forming impure computations, the semantics is deterministic and strongly normalizing: given any \( M : \mathcal{C} \), 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 ground type \( G \) is a value type that does not contain thunks:

\[
G ::= \text{unit} \mid G_1 \times G_2 \mid \text{empty} \mid G_1 + G_2
\]

A CBPV program is a closed computation \( M : \mathbf{F}G \), where \( G \) is a ground type. 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 \( \preceq \) on closed computations of type \( G \), for each ground type \( G \). (We leave \( G \) implicit when writing \( \preceq \).) For example, we could use

\[
M \preceq M' \text{ if and only if } \forall V: \text{bool}. (M \Downarrow \text{return } V) \Rightarrow (M' \Downarrow \text{return } V)
\]

We could also use, for example, the indiscrete relation for \( \preceq \) (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 reasoning principle would be trivial).

---

1 We do not actually need to assume that \( \preceq \) 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 \( \preceq \) is not a preorder.
Galois connecting call-by-value and call-by-name

1.'\lambda\{1.\ M_1,2.\ M_2\} \equiv M_1 \quad \text{if } true \text{ then } M_1 \text{ else } M_2 \equiv M_1
2.'\lambda\{1.\ M_1,2.\ M_2\} \equiv M_2 \quad \text{if } false \text{ then } M_1 \text{ else } M_2 \equiv M_2
V.'\lambda x:\ A.\ M \equiv M[x \mapsto V] \quad \text{force thunk } M \equiv M
match (V_1,V_2) \text{ with } (x_1,x_2).\ M \equiv M[x_1 \mapsto V_1,x_2 \mapsto V_2]

V \equiv () \quad V \equiv \text{ thunk force } V
M[x \mapsto V] \equiv \text{ match } V \text{ with } (x_1,x_2).\ M[x \mapsto (x_1,x_2)] \quad M \equiv \lambda\{1.1'\ M,2.2'\ M\}
M[x \mapsto V] \equiv \text{ if } V \text{ then } M[x \mapsto \text{ true}] \text{ else } M[x \mapsto \text{ false}] \quad M \equiv \lambda x:A.x'xM
\text{return } V \text{ to } x.\ M \equiv M[x \mapsto V] \quad M \equiv M \text{ to } x.\ \text{return } x
(M_1 \text{ to } x.\ M_2) \text{ to } y.\ M_3 \equiv M_1 \text{ to } x.\ (M_2 \text{ to } y.\ M_3)
\lambda\{1.\ M \text{ to } x.\ N_1,2.\ M \text{ to } x.\ N_2\} \equiv M \text{ to } x.\ \lambda\{1.\ N_1,2.\ N_2\}
\lambda y:A.\ M \text{ to } x.\ N \equiv M \text{ to } x.\ \lambda y:A.\ N

\text{Figure 3} \text{ (Typed) equations between CBPV terms}

Given any program relation \(\preceq\), we define a contextual preorder \(M \preceq_{\text{ctx}}^\Gamma M'\) on arbitrary well-typed computations (in typing context \(\Gamma\)) by considering the behaviour of \(M\) and \(M'\) in programs as follows. A computation context \(E\) is a computation term, with a single hole \(\Box\) where a computation term is expected. We write \(E[M]\) for the computation that results from replacing \(\Box\) with \(M\) (which may capture some of the free variables of \(M\)). For example, if \(E\) is the computation context \(N \text{ to } x.\ \Box\) then \(E[\text{return } x]\) is the computation \(N \text{ to } x.\ \text{return } x\), where \(x\) is captured. We use computation contexts to define \(\preceq_{\text{ctx}}^\Gamma\).

\textbf{Definition 1 (Contextual preorder).} Suppose that \(\preceq\) is a program relation, and that \(\Gamma \vdash_c C\) and \(\Gamma \vdash_c C'\) are two computations of the same type. We write \(M \preceq_{\text{ctx}}^\Gamma M'\) if, for all ground types \(G\) and computation contexts \(E\) such that \(E[M]\), \(E[M']\) : \(FG\), we have \(E[M] \preceq E[M']\).

We sometimes omit \(\Gamma\), and write just \(M \preceq_{\text{ctx}} M'\).

In parts of this paper, we use an equational theory on terms. We write \(\equiv\) for the smallest equivalence relation on terms of the same type that is closed under the axioms in Figure 3 and under the syntax of CBPV terms (for example, \(M \equiv N\) implies \(\text{thunk } M \equiv \text{ thunk } N\) and \(V \equiv W\) implies \(\text{return } V \equiv \text{ return } W\)).\(^2\) All of the axioms should be read as subject to suitable typing constraints. In particular, \(V \equiv ()\), which is the \(\eta\)-law for \text{unit}, can only be applied if \(V\) has type ()\(^.\) The group of axioms at the top of Figure 3 contains the \(\beta\)-laws for all of the type formers except \text{F}. The second group contains \(\eta\)-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 forms for pairs of computations and for functions. If two computations are related by \(\equiv\) then they behave the same under evaluation. In particular, at returner types:

\(^2\) This is not exactly Levy’s equational theory for CBPV, because we do not include complex values.
Lemma 2. If \( M, N : F A \) are closed computations that satisfy \( M \equiv N \), and \( M \Downarrow \text{return}\ V \), then there is some \( W : A \) such that \( N \Downarrow \text{return}\ W \) and \( V \equiv W \).

The proof of this uses a bisimulation argument (see Pitts [21]).

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 [13] 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 \( \tau \) and expressions \( e \):

\[
\tau ::= \text{unit} | \tau_1 \times \tau_2 | \tau_1 \to \tau_2' \\
e ::= x | () | (e_1, e_2) | \text{fst}\ e | \text{snd}\ e | \text{true} | \text{false} | \text{if}\ e_0\ \text{then}\ e_1\ \text{else}\ e_2 | \lambda x:\tau.e | e e'
\]

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 4. For call-by-value, each source language type \( \tau \) maps to a CBPV value type \( L\tau M^v \) that contains the results of call-by-value computations. For call-by-name, \( \tau \) is translated to a computation type \( L\tau M^n \), which contains the computations themselves. Products in call-by-value use the value-type products of CBPV (which means they are necessarily \textit{strict}: both components of a pair are always evaluated). For call-by-name we give a lazy interpretation of binary products, using products of CBPV computation types. (Though note that we do not interpret \text{unit} as a nullary product of computation types.) Functions under the call-by-value translation accept values of type \( L\tau M^v \) 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 \( \Gamma \) are translated into CBPV typing contexts \( L\Gamma M^v \) and \( L\Gamma M^n \). 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 \( L e M^v \) and \( L e M^n \). The translation uses some auxiliary program variables, which are assumed fresh.

For call-by-value we arbitrarily choose left-to-right evaluation for both pairing and function application. Under the call-by-name translation, side-effects occur only at the base types \text{unit} and \text{bool} (since this is where the returner types appear). As a consequence, \( \eta \)-expansion of source-language expressions is semantics-preserving: \( \langle e \rangle^v \equiv \langle \lambda x : A. e x \rangle^v \) for functions and \( \langle e \rangle^n \equiv \langle (\text{fst}\ e, \text{snd}\ e) \rangle^n \) for binary products. This is not the case for call-by-value.

Of course, we have to justify that these translations actually capture call-by-value and call-by-name. There are two semantics of interest for the source language: a call-by-value semantics (that evaluates left-to-right), and a call-by-name semantics (with lazy products). Since we consider the observable behaviour of CBPV terms, the properties we want are that if the call-by-value translations \( \langle e \rangle^v \) and \( \langle e' \rangle^v \) have the same observable behaviour then \( e \) and \( e' \) have the same observable behaviour with respect to the call-by-value semantics, and similarly for call-by-name. Levy [13] proves both of these properties (though without products in the source language). We take this as the required justification, and do not give the details.

2.2 Examples

We consider three collections of (allowable) side-effects as examples throughout the paper.
### Call-by-Value Translation

**Types**

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\langle \tau \rangle^v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>unit</td>
</tr>
<tr>
<td>$\tau_1 \times \tau_2$</td>
<td>$\langle \tau_1 \rangle^v \times \langle \tau_2 \rangle^v$</td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
</tr>
<tr>
<td>$\tau \rightarrow \tau'$</td>
<td>$U \langle \tau \rangle^v \rightarrow \mathbf{F} \langle \tau' \rangle^v$</td>
</tr>
</tbody>
</table>

**Typing Contexts**

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$\langle \Gamma \rangle^v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : \tau$</td>
<td>$\langle \Gamma \rangle^v, x : \langle \tau \rangle^v$</td>
</tr>
</tbody>
</table>

**Expressions**

<table>
<thead>
<tr>
<th>$\Gamma \vdash e : \tau$</th>
<th>$\langle \langle \Gamma \rangle^v \vdash_e \langle e \rangle^v : \mathbf{F} \langle \tau \rangle^v \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\text{return} x$</td>
</tr>
<tr>
<td>()</td>
<td>$\text{return} ()$</td>
</tr>
<tr>
<td>$(e_1, e_2)$</td>
<td>$\langle e_1 \rangle^v \text{ to } z_1. \langle e_2 \rangle^v \text{ to } z_2. \text{return} (z_1, z_2)$</td>
</tr>
<tr>
<td>$\text{fst } e$</td>
<td>$\langle e \rangle^v \text{ to } z. \text{match } z \text{ with } (z_1, z_2). \text{return } z_1$</td>
</tr>
<tr>
<td>$\text{snd } e$</td>
<td>$\langle e \rangle^v \text{ to } z. \text{match } z \text{ with } (z_1, z_2). \text{return } z_2$</td>
</tr>
<tr>
<td>true</td>
<td>$\text{return } \text{true}$</td>
</tr>
<tr>
<td>false</td>
<td>$\text{return } \text{false}$</td>
</tr>
<tr>
<td>if $e_0$ then $e_1$ else $e_2$</td>
<td>$\langle e_0 \rangle^v \text{ to } z. \text{if } z \text{ then } \langle e_1 \rangle^v \text{ else } \langle e_2 \rangle^v$</td>
</tr>
<tr>
<td>$\lambda x : \tau. e$</td>
<td>$\text{return thunk } \lambda x : \langle \tau \rangle^v. \langle e \rangle^v$</td>
</tr>
<tr>
<td>ee'</td>
<td>$\langle e \rangle^v \text{ to } y. \langle e' \rangle^v \text{ to } z. z' \text{ force } y$</td>
</tr>
</tbody>
</table>

(a) Call-by-value translation $\langle - \rangle^v$

### Call-by-Name Translation

**Types**

<table>
<thead>
<tr>
<th>$\tau$</th>
<th>$\langle \tau \rangle^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit</td>
<td>$\mathbf{F} \text{unit}$</td>
</tr>
<tr>
<td>$\tau_1 \times \tau_2$</td>
<td>$\langle \tau_1 \rangle^n \times \langle \tau_2 \rangle^n$</td>
</tr>
<tr>
<td>bool</td>
<td>$\mathbf{F} \text{bool}$</td>
</tr>
<tr>
<td>$\tau \rightarrow \tau'$</td>
<td>$U \langle \tau \rangle^n \rightarrow \langle \tau' \rangle^n$</td>
</tr>
</tbody>
</table>

**Typing Contexts**

<table>
<thead>
<tr>
<th>$\Gamma$</th>
<th>$\langle \Gamma \rangle^n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma, x : \tau$</td>
<td>$\langle \Gamma \rangle^n, x : U \langle \tau \rangle^n$</td>
</tr>
</tbody>
</table>

**Expressions**

<table>
<thead>
<tr>
<th>$\Gamma \vdash e : \tau$</th>
<th>$\langle \langle \Gamma \rangle^n \vdash_e \langle e \rangle^n : \langle \tau \rangle^n \rangle$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\text{force } x$</td>
</tr>
<tr>
<td>()</td>
<td>$\text{return} ()$</td>
</tr>
<tr>
<td>$(e_1, e_2)$</td>
<td>$\lambda { 1. \langle e_1 \rangle^n, 2. \langle e_2 \rangle^n }$</td>
</tr>
<tr>
<td>$\text{fst } e$</td>
<td>$1. \langle e \rangle^n$</td>
</tr>
<tr>
<td>$\text{snd } e$</td>
<td>$2. \langle e \rangle^n$</td>
</tr>
<tr>
<td>true</td>
<td>$\text{return } \text{true}$</td>
</tr>
<tr>
<td>false</td>
<td>$\text{return } \text{false}$</td>
</tr>
<tr>
<td>if $e_0$ then $e_1$ else $e_2$</td>
<td>$\langle e_0 \rangle^n \text{ to } z. \text{if } z \text{ then } \langle e_1 \rangle^n \text{ else } \langle e_2 \rangle^n$</td>
</tr>
<tr>
<td>$\lambda x : \tau. e$</td>
<td>$\lambda x : U \langle \tau \rangle^n. \langle e \rangle^n$</td>
</tr>
<tr>
<td>ee'</td>
<td>$(\text{thunk } \langle e' \rangle^n) \cdot \langle e \rangle^n$</td>
</tr>
</tbody>
</table>

(b) Call-by-name translation $\langle - \rangle^n$

---

**Figure 4** Translations from the source language into CBPV
Example 3 (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 \( M \leq M' \) (for closed computations \( M, M' : F\)bool) as:

\[
M \leq M' \text{ if and only if } \exists V : \text{bool}. (M \Downarrow \text{return} V) \land (M' \Downarrow \text{return} V)
\]

In other words, \( M \) and \( M' \) both evaluate to the same result \( V \). Since evaluation is deterministic, \( V \) is necessarily unique. The contextual preorder \( M \leq_{ctx} M' \) 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 4 (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 \( \leq_{ctx} \) 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 \( \text{rec} \text{x:} \text{U}C. M \), and correspondingly extend the type system and operational semantics with the following rules:

\[
\begin{align*}
\Gamma, \text{x:} \text{U} C & \vdash M : C \\
\Gamma & \vdash \text{rec} \text{x:} \text{U} C. M : C \\
M'[x \mapsto \text{thunk}(\text{rec} x : \text{U} C. M)] & \Downarrow R
\end{align*}
\]

The variable \( x \) is bound to a thunk of the recursive computation, so recursion is done by forcing \( x \). (This is not the only way to add recursion to CBPV [3], but is the most convenient for our purposes.) Of course, by adding recursion we lose strong normalization (but the semantics is still deterministic). We extend the source language, and the two translations into CBPV, with recursive functions:

\[
\begin{align*}
e & ::= \ldots | \text{rec} f : \tau \rightarrow \tau'. \lambda x. e & \Gamma, f : \tau \rightarrow \tau', x : \tau \vdash e : \tau' \\
\Gamma & \vdash \text{rec} f : \tau \rightarrow \tau'. \lambda x. e : \tau \rightarrow \tau'
\end{align*}
\]

\[
\begin{align*}
\langle \text{rec} f : \tau \rightarrow \tau'. \lambda x. e \rangle^V & = \text{return} \text{thunk} \langle \text{rec} f : \text{U} \langle \tau \rangle^V \rightarrow \text{F} \langle \tau' \rangle^V, \lambda x : \langle \tau \rangle^V. \langle e \rangle^V \rangle \\
\langle \text{rec} f : \tau \rightarrow \tau'. \lambda x. e \rangle^n & = \text{rec} f : \text{U} \langle \tau \rangle^n \rightarrow \langle \tau' \rangle^n, \lambda x : \langle \tau \rangle^n, \langle e \rangle^n
\end{align*}
\]

Again, the translations are the same as those given by Levy [13], except that Levy has general fixed points for call-by-name, rather than just recursive functions. The expression \( \Omega_\tau = \langle (\text{rec} f : \text{bool} \rightarrow \tau. \text{x}. f x) \text{false} \rangle : \tau \) enables us to distinguish between call-by-value and call-by-name: \( (\lambda x : \tau. \text{false}) \Omega_\tau \) diverges in call-by-value but not in call-by-name. In particular, we have \( \langle (\lambda x : \tau. \text{true}) \Omega_\tau \rangle^n \Downarrow \text{return} \text{true} \), but there is no \( R \) such that \( \langle (\lambda x : \tau. \text{true}) \Omega_\tau \rangle^\tau \Downarrow R \).

For this example, we define the program relation \( \leq \) by

\[
M \leq M' \text{ if and only if } \forall V : \text{bool}. (M \Downarrow \text{return} V) \Rightarrow (M' \Downarrow \text{return} V)
\]

so that \( M \leq_{ctx} M' \) 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 5 (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 \texttt{fail}_C for nullary nondeterministic choice and \texttt{M or N} for binary nondeterministic choice between computations; the typing and evaluation rules are standard:

\[
\begin{array}{c c c}
\Gamma \vdash e : \tau & \Gamma \vdash e' : \tau & (\text{fail}_\tau)^\gamma = \text{fail}_{\tau^\gamma} \\
\Gamma \vdash \text{fail}_C : C & \Gamma \vdash e \text{ or } e' : \tau & (e \text{ or } e')^\gamma = (e)^\gamma \text{ or } (e')^\gamma \\
& \text{ or } N : C & (\text{fail}_C \text{ or } N)^\gamma = (\text{fail}_C)^\gamma \text{ or } (N)^\gamma
\end{array}
\]

(There is no \( R \) such that \( \text{fail}_C \downarrow R \).) The computation \( \text{fail}_C \) is the unit for \texttt{or}, so \texttt{M or fail}_C and \texttt{M or fail}_C have the same behaviour as \( M \). For each closed computation \( M : F A \) there might be zero, one or several values \( V : A \) such that \( M \downarrow \text{return } V \).

We similarly include nullary and binary nondeterminism in the source language, and extend the call-by-value and call-by-name translations:

\[
\begin{array}{c c c}
\Gamma \vdash e : \tau & \Gamma \vdash e' : \tau & (\text{fail}_\tau)^\gamma = \text{fail}_{\tau^\gamma} \\
\Gamma \vdash \text{fail}_C : C & \Gamma \vdash e \text{ or } e' : \tau & (e \text{ or } e')^\gamma = (e)^\gamma \text{ or } (e')^\gamma \\
& \text{ or } N : C & (\text{fail}_C \text{ or } N)^\gamma = (\text{fail}_C)^\gamma \text{ or } (N)^\gamma
\end{array}
\]

As an example, evaluating the expression \( e = (\lambda x. \text{if } x \text{ then } x \text{ else true})(\text{true or false}) \) under call-by-value necessarily results in \texttt{true}, but under call-by-name we can also get \texttt{false}. (We have \( (\text{true})^\gamma \not\downarrow \text{return false} \) but \( (\text{false})^\gamma \downarrow \text{return false} \).)

For nondeterminism, we define \( \lll \) in the same way as our divergence example:

\[
M \lll M' \text{ if and only if } \forall V : \text{bool}. (M \downarrow \text{return } V) \Rightarrow (M' \downarrow \text{return } V)
\]

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 \( \lll_{\text{ctx}} \). Since \( \lll_{\text{ctx}} \) 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 \( M \lll_{\text{ctx}} N \). Our semantics is based on Levy’s algebra models [15] 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 [14] can be used for the same purpose.)

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

\begin{itemize}
\item \textbf{Definition 6}. A \textbf{Poset}-category \( C \) is an ordinary category, together with a partial order \( \sqsubseteq \) on each hom-set \( C(X,Y) \), such that composition is monotone.
\end{itemize}

If \( C \) is a \textbf{Poset}-category, we refer to the ordinary category as the \textit{underlying} ordinary category, and write \([C]\) for the class of objects.

\begin{itemize}
\item \textbf{Example 7}. We use the following three \textbf{Poset}-categories.
\end{itemize}

| Poset-category \( C \) | Objects \( X \in |C| \) | Morphisms \( f : X \to Y \) | Order \( f \sqsubseteq f' \) |
|-------------------------|-----------------------|------------------|------------------|
| Set                     | sets                  | functions        | equality         |
| Poset                   | posets                | monotone functions | pointwise         |
| \( \omega \text{Cpo} \) | \( \omega \text{pos} \) | \( \omega \text{-continuous functions} \) | pointwise         |
In each case, composition and identities are defined in usual way. For \textbf{Set}, since the hom-sets \textbf{Set}(_X,Y_) are discrete, all of the \textbf{Poset}-enriched definitions coincide with the ordinary (unenriched) definitions. The objects of \omega\textbf{Cpo} are posets (_X,\sqsubseteq_) for which \sqsubseteq is \omega-complete, i.e. for which every \omega-chain \{x_0 \sqsubseteq x_1 \sqsubseteq \ldots\} has a least upper bound \bigcup x_i. Morphisms are \omega-continuous functions, i.e. monotone functions that preserve least upper bounds of \omega-chains.

Let \textbf{C} be a \textbf{Poset}-category. We say that \textbf{C} is cartesian when its underlying category has a terminal object 1 and binary products _X_ \times _Y_ , such that the pairing functions \langle _,_\rangle : \textbf{C}(\textit{W},_X_ \times _Y_) \times \textbf{C}(\textit{W},_X_ \times _Y_) \to \textbf{C}(\textit{W},_X_ \times _Y_) are monotone. When this is the case, there are canonical isomorphisms \text{assoc}_{_X_ \times _Y_ \times _X_} : \textbf{C}(_X_ \times _X_ \times _Y_,_X_ \times _Y_) \to \textbf{C}(_X_ \times _X_ \times _Y_,_X_ \times _Y_). We say that \textbf{C} is cartesian closed when it is cartesian and its underlying category has exponentials \textit{X} \Rightarrow \textit{Y} for which the currying functions \text{A} : \textbf{C}(\textit{W} \times _X_ \times _Y_) \to \textbf{C}(\textit{W} \times _X_ \Rightarrow _Y_) are monotone. (It follows that the uncurrying functions \text{A}^{-1} : \textbf{C}(\textit{W},_X_ \Rightarrow _Y_) \to \textbf{C}(\textit{W} \times _X_ \times _Y_) are also monotone.) We write \text{ev}_{_X_ \times _Y_} for the canonical morphism \text{A}^{-1}(\text{id}) : (_X_ \Rightarrow _Y_) \times _X_ \times _Y_ \to _Y_. Binary coproducts in \textbf{C} are just binary coproducts in the underlying ordinary category, except that the copairing functions \langle _,_\rangle : \textbf{C}(\textit{W},_X_ \times _Y_) \times \textbf{C}(\textit{W},_X_ \times _Y_) \to \textbf{C}(\textit{W},_X_ \times _Y_) are required to be monotone. The \textbf{Poset}-categories \textbf{Set}, \textbf{Poset}, and \omega\textbf{Cpo} are all cartesian closed, and have binary coproducts.

We interpret computation types as (Eilenberg-Moore) algebras for an order-enriched monad \textit{T}, which we need to be strong (just as models of Moggi’s monadic metalanguage [20] use a strong monad). The definitions of strong \textbf{Poset}-monad and of \textit{T}-algebra we give are slightly non-standard, but are equivalent to the standard ones. (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.)

\begin{definition}[Strong Poset-monad] A strong \textbf{Poset}-monad \textit{T} = (\textit{T}, \eta, (_-)^\dagger) on a cartesian \textbf{Poset}-category \textbf{C} consists of an object \textit{TX} \in |\textbf{C}| and morphism \eta_{_X} : _X_ \to \textit{TX} for each \textit{X} \in |\textbf{C}|, and a monotone function (Kleisli extension) \langle _-,_-_\rangle : \textbf{C}(\textit{W} \times _X_ \times _Y_,_Y_) \to \textbf{C}(\textit{W} \times \textit{TX},_Y_) for each \textit{W},_X_ \times _Y_ \in |\textbf{C}|, such that

\begin{align*}
(f^\dagger \circ (W \times \eta_{_X})) &= f : \textit{W} \times _X_ \to \textit{TY} & \text{for all } f : \textit{W} \times _X_ \to \textit{TY} \\
((\eta_{_X} \circ \pi_2)^\dagger) &= \pi_2 : 1 \times \textit{TX} \to \textit{TX} & \text{for all } \textit{X} \in |\textbf{C}| \\
g^\dagger \circ (W' \times f) \circ \text{assoc}^\dagger &= g^\dagger \circ (W' \times f^\dagger) \circ \text{assoc} & \text{for all } f : \textit{W} \times _X_ \to _Y_, \\
& : (W' \times \textit{W}) \times \textit{TX} \to \textit{TZ} & g : \textit{W}' \times _Y_ \to \textit{TZ}
\end{align*}

Specializing the Kleisli extension of \textit{T} to \textit{W} = 1 produces a (non-strong) extension operator \langle _-,_-_\rangle : \textbf{C}(\textit{X},_Y_) \to \textbf{C}(\textit{TX},_Y_). We use this to define, for every \textit{f} : _X_ \to _Y_, a morphism \textit{Tf} : \textit{TX} \to \textit{TY} by \textit{Tf} = (\eta_{_Y} \circ f)^\dagger. (The latter definition makes \textit{T} into a \textbf{Poset}- functor.)

\begin{definition}[Eilenberg-Moore algebra] Let \textit{T} be a strong \textbf{Poset}-monad on a cartesian \textbf{Poset}-category \textbf{C}. A \textit{T}-algebra \textit{Z} = (\textit{Z}, (_-)^\dagger) is a pair of an object \textit{Z} \in |\textbf{C}| (the carrier) and monotone function (extension operator) \langle _-,_-_\rangle : \textbf{C}(\textit{W} \times _X_ \times _Z_,_Z_) \to \textbf{C}(\textit{W} \times \textit{TX},_Z_) for each \textit{W},_X_ \times _Z_ \in |\textbf{C}|, such that

\begin{align*}
(f^\dagger \circ (W \times \eta_{_X})) &= f : \textit{W} \times _X_ \to _Z & \text{for all } f : \textit{W} \times _X_ \to _Z \\
((\eta_{_X} \circ \pi_2)^\dagger) &= \pi_2 : 1 \times \textit{TX} \to \textit{TX} & \text{for all } \textit{X} \in |\textbf{C}| \\
g^\dagger \circ (W' \times f) \circ \text{assoc}^\dagger &= g^\dagger \circ (W' \times f^\dagger) \circ \text{assoc} & \text{for all } f : \textit{W} \times _X_ \to _Y_, \\
& : (W' \times \textit{W}) \times \textit{TX} \to _Z & g : \textit{W}' \times _Y_ \to _Z
\end{align*}

For each \textit{X} \in |\textbf{C}|, we write \textit{FTX} for the free \textit{T}-algebra (\textit{TX}, (_-)^\dagger), and for each \textit{T}-algebra \textit{Z}, we write \textit{UTZ} for the carrier \textit{Z} \in |\textbf{C}|.

\end{definition}
Galois connecting call-by-value and call-by-name

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

Let $T$ be a strong Poset-monad on a cartesian closed Poset-category $C$. If $Z_1, Z_2$ are $T$-algebras, then there is a $T$-algebra $Z_1 \times Z_2$ with carrier $Z_1 \times Z_2$ and extension operator

$$f^! = ((\pi_1 \circ f)^!, (\pi_2 \circ f)^!): W \times TX \to Z_1 \times Z_2$$

(These are binary products in the Eilenberg-Moore Poset-category of $T$.) If $Y \in [C]$ and $Z$ is a $T$-algebra, then there is a $T$-algebra $Y \Rightarrow Z$ with carrier $Y \Rightarrow Z$ and extension operator

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

where $\beta_{X_1,X_2,X_3} = ((\pi_1 \circ \pi_1, \pi_2 \circ \pi_1) : (X_1 \times X_2) \times X_3 \to (X_1 \times X_3) \times X_2$. These provide the interpretations of binary products $\mathbb{C}_1 \times \mathbb{C}_2$ of computation types, and the interpretations of function types $A \to C$.

**Definition 10.** A model of CBPV consists of
- a cartesian closed Poset-category $C$;
- an object $2$ and morphisms $\text{inl}, \text{inr} : 1 \to 2$, forming the coproduct $1 + 1$;
- a strong Poset-monad $T = (T, \eta, (-)^!)$ on $C$.

Given any model, the interpretation $\llbracket - \rrbracket$ of CBPV is defined in Figure 5. Value types $A$ are interpreted as objects $[A] \in [C]$, while computation types $C$ are interpreted as $T$-algebras. Typing contexts $\Gamma$ are interpreted as objects $[\Gamma] \in C$ using the cartesian structure of $C$; if $(x : A) \in \Gamma$ then we write $\pi_x$ for the corresponding projection $[\Gamma] \to [A]$. Values $\Gamma \vdash V : A$ (respectively computations $\Gamma \vdash_c M : C$) are interpreted as morphisms $[\Gamma \vdash V : A]$ (resp. $[\Gamma \vdash_c M : C]$) in $C$; we often omit the typing context and type when writing these. Programs $\sigma_{\vdash_c} M : \text{bool}$ are therefore interpreted as morphisms $[M] : 1 \to \mathbb{P}_2$. To interpret if, we use the fact that, since $C$ is cartesian closed, products distribute over the coproduct $2 = 1 + 1$. This means that for every $W \in [C]$, the coproduct $W + W$ also exists in $C$, and the canonical morphism

$$W + W \overset{[(\text{id}_W, \text{inl}_W), (\text{id}_W, \text{inr}_W)]}{\longrightarrow} W \times 2$$

has an inverse $\text{dist}_W : W \times 2 \to W + W$.

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

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

**Definition 11.** A model of CBPV is adequate (with respect to a given program relation $\approx$) if for all computations $\Gamma \vdash_c M : C$ and $\Gamma \vdash_c M' : C$ we have

$$[\Gamma \vdash_c M : C] \subseteq [\Gamma \vdash_c M' : C] \quad \Rightarrow \quad M \approx_{\text{ctx}} M'$$

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 $\approx$; the proof in each case is a standard logical relations argument (e.g. [28]).

**Example 12.** For CBPV with no side-effects, we use $C = \text{Set}$. The strong Poset-monad $T$ is the identity on Set. Each $T$-algebra $Z$ is completely determined by its carrier $Z$; the extension operator $(-)^! : \text{Set}(W \times X, Z) \to \text{Set}(W \times X, Z)$ is necessarily the identity. The interpretation $[M]$ of each program $M$ is just an element of 2.
\begin{align*}
\text{[C-object [A]]} \\
\text{[unit]} &= 1 \\
\text{[A_1 \times A_2]} &= [A_1] \times [A_2] \\
\text{[bool]} &= 2 \ (= 1+1) \\
\text{[U C]} &= U_T[C] \\
\text{[\Gamma \vdash V : [\Gamma] \rightarrow [A]]} \quad [x] &= \pi_x \\
\text{[\not\exists]} &= \langle \rangle_{[\Gamma]} \\
\text{[(V_1, V_2)]} &= ([V_1], [V_2]) \\
\text{[true]} &= \text{inl} \circ \langle \rangle_{[\Gamma]} \\
\text{[false]} &= \text{inr} \circ \langle \rangle_{[\Gamma]} \\
\text{[thunk M]} &= [M] \\
\text{[T-algebra [C]]} \\
\text{[C_1 \times C_2]} &= [C_1] \times [C_2] \\
\text{[A \rightarrow C]} &= [A] \Rightarrow [C] \\
\text{[F A]} &= F_T[A] \\
\text{[\Gamma \vdash M : [\Gamma] \rightarrow U_T[C]]} \\
\text{[\lambda \{1. M_1, 2. M_2\}]} &= \langle [M_1], [M_2] \rangle \\
\text{[i. M]} &= \pi_i \circ [M] \quad (i \in \{1, 2\}) \\
\text{[\lambda x : A. M]} &= \Lambda [M] \\
\text{[V^{\cdot} M]} &= \Lambda^{-1} [M] \circ \langle id, [V] \rangle \\
\text{[\text{return} V]} &= \eta \circ [V] \\
\text{[\text{match} V \text{ with} (x, y). M]} &= [M] \circ \text{assoc}^{-1} \circ \langle id, [V] \rangle \\
\text{[\text{if} V \text{ then} M_1 \text{ else} M_2]} &= [[M_1], [M_2]] \circ \text{dist} \circ \langle id, [V] \rangle \\
\text{[\text{force} V]} &= [V]
\end{align*}

\textbf{Figure 5} Denotational semantics of CBPV
Example 13. For divergence, we use $C = \omega \text{Cpo}$. The strong Poset-monad $T$ freely adjoins a least element $\bot$ to each $\omega \text{Cpo}$. The unit $\eta X$ 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 $\omega \text{Cpo}$ $Z$ with a least element $\bot \in Z$. The extension operator is completely determined once the carrier is fixed; it is analogous to $(-)^\dagger$.

In this case, the product $Z_1 \times Z_2$ is the set of pairs ordered componentwise, and the exponential $Y \Rightarrow Z$ is the set of set of $\omega$-continuous functions ordered pointwise. Hence $Z_1 \times Z_2$ has a least element $(\bot, \bot)$ (so forms a $T$-algebra) whenever $Z_1$ and $Z_2$ have least elements, and $Y \Rightarrow Z$ has a least element (the constantly-$\bot$ function) whenever $Z$ has a least element.

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

Example 14. For finite nondeterminism, we use $C = \text{Poset}$. The strong Poset-monad $T$ freely adds finite joins to each poset. It is defined by

$$TX = (\downarrow S' \mid S' \in P_{\text{fin}} X), \sqsubseteq \quad \eta x = \downarrow \{x\} \quad f^\dagger (w, S) = \bigsqcup_{x \in S} f(w, x)$$

where $P_{\text{fin}} X$ 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 $f^\dagger (w, S) = \bigsqcup_{x \in S} f(w, x)$. (The latter join exists because $S$ is the downwards-closure of a finite set, even though $S$ itself might not be finite.) The product $Z_1 \times Z_2$ is the set of pairs ordered componentwise, with joins given by $\bigsqcup_{i} (z_i, z'_i) = (\bigsqcup_{i} z_i, \bigsqcup_{i} z'_i)$. The function space $Y \Rightarrow Z$ is the set of monotone functions ordered pointwise, with joins given by $(\bigsqcup_{i} f_i) x = \bigsqcup_{i} (f_i x)$.

We interpret nondeterministic computations using nullary and binary joins:

$$[\text{fail}] \rho = \bot \quad [M \text{ or } N] \rho = [M] \rho \cup [N] \rho$$

The interpretation $[M]$ of a program $M : \text{Fbool}$ is one of the four subsets of 2.

4 Left-to-right or right-to-left?

Before we consider call-by-value and call-by-name, we consider a simpler case of a relationship between evaluation orders. We show that, under certain conditions (the assumptions of Theorem 15 below), left-to-right evaluation of pairs is equivalent to right-to-left evaluation in call-by-value. (We could also consider right-to-left evaluation of function application with no extra difficulty.)

The call-by-value translation $\langle e \rangle^\nu$ defined in Figure 4 evaluates pairs from left-to-right. We give another call-by-value translation $\langle - \rangle^\nu$, which has an identical definition to $\langle - \rangle^\nu$ except that pairs are evaluated right-to-left:

$$\langle (e_1, e_2) \rangle^\nu = \langle e_2 \rangle^\nu \text{ to } z_2, \langle e_1 \rangle^\nu \text{ to } z_1. \text{return } (z_1, z_2)$$
The left-to-right and right-to-left translations have the same types, and therefore it makes sense to ask whether \( \langle e \rangle^\tau \cong \langle e \rangle^\upsilon \) holds, where \( M \cong_{\text{ctx}} M' \) means both \( M \cong_{\text{ctx}} M' \) and \( M' \cong_{\text{ctx}} M \) hold. (Relating call-by-value and call-by-name is more difficult because the two translations have different types.)

Since we reason using the denotational semantics, we first spell out the interpretations of these two translations of pairs given any CBPV model. We define two natural transformations for sequencing of computations: \( \text{seq} \) for left-to-right and \( \text{seq}' \) for right-to-left. These are defined in terms of a monotone function \( (-)^\tau : C(X \times W, Y) \to C(TX \times W, Y) \) (which is Kleisli extension with the products reversed):

\[
\text{seq}_{X_1, X_2} = (\eta_{X_1 \times X_2})^\tau : TX_1 \times TX_2 \to T(X_1 \times X_2)
\]

\[
\text{seq}'_{X_1, X_2} = (\eta_{X_1 \times X_2})^\upsilon : TX_1 \times TX_2 \to T(X_1 \times X_2)
\]

In our three example models, the natural transformation \( \text{seq} \) is given by:

\[
\text{seq}_{X_1, X_2}(x_1, x_2) = (x_1, x_2) \quad \text{(no side-effects)}
\]

\[
\text{seq}_{X_1, X_2}(x_1, x_2) = \begin{cases} 
\perp & \text{if } x_1 = \perp \lor x_2 = \perp \\
(x_1, x_2) & \text{otherwise}
\end{cases} \quad \text{(divergence)}
\]

\[
\text{seq}_{X_1, X_2}(S_1, S_2) = \{(x_1, x_2) \mid x_1 \in S_1 \land x_2 \in S_2\} \quad \text{(nondeterminism)}
\]

Writing \( [[-]]^\tau \) for \( [[-]]^\upsilon \) and \( [[-]]^\tau \) for \( [[-]]^\upsilon \), we have:

\[
[[e_1, e_2]]^\tau = \text{seq} \circ [[e_1]]^\upsilon, [[e_2]]^\upsilon \quad [[e_1, e_2]]^\upsilon = \text{seq}' \circ [[e_1]]^\tau, [[e_2]]^\tau
\]

We do not have \( \langle e \rangle^\upsilon \cong_{\text{ctx}} \langle e \rangle^\tau \) in general, because side-effects might occur in (observably) different orders. However, we do have \( \langle e \rangle^\upsilon \cong_{\text{ctx}} \langle e \rangle^\tau \) for our three examples (no side-effects, divergence and nondeterminism). The crucial property is that in all three cases, the monad \( T \) is commutative \([9]\), meaning that \( \text{seq} = \text{seq}' \). Commutativity implies \( \langle e \rangle^\upsilon = \langle e \rangle^\tau \), and then adequacy implies \( \langle e \rangle^\upsilon \cong_{\text{ctx}} \langle e \rangle^\tau \). Hence we arrive at a reasoning principle for relating left-to-right and right-to-left evaluation. Given a program relation \( \prec \), to show \( \langle e \rangle^\upsilon \cong_{\text{ctx}} \langle e \rangle^\tau \), it suffices to find an adequate model that satisfies commutativity:

**Theorem 15** (Relationship between left-to-right and right-to-left). Suppose we are given a program relation \( \prec \), and a model of CBPV that is adequate with respect to \( \prec \) and satisfies \( \text{seq} = \text{seq}' \). If \( \Gamma \vdash e : \tau \) then \( \langle e \rangle^\upsilon \cong_{\text{ctx}} \langle e \rangle^\tau \).

**Proof.** By induction on \( e \). The only non-trivial case is pairing, for which we use \( \text{seq} = \text{seq}' \).

The key property required for the proof, which follows from commutativity, is centrality:

**Definition 16** (Central \([23]\)). A computation \( \Gamma \vdash e : F A \) is central if for all computations \( \Gamma \vdash e : N : F B \) we have

\[
\text{seq} \circ \langle f, g \rangle = \text{seq}' \circ \langle f, g \rangle
\]

A morphism \( f : Z \to TX \) is central if for all \( g : Z \to TY \) we have

\[
\text{seq} \circ \langle f, g \rangle = \text{seq}' \circ \langle f, g \rangle
\]
5 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. This is more difficult than relating left-to-right and right-to-left evaluation, because the call-by-value and call-by-name translations of expressions have different types:

\[ \langle \Gamma \rangle^V \vdash_c \langle e \rangle^V : F \langle \tau \rangle^V \quad \langle \Gamma \rangle^n \vdash_c \langle e \rangle^n : \langle \tau \rangle^n \]

We cannot ask whether \( \langle e \rangle^V \) and \( \langle e \rangle^n \) are related by \( \preceq_{\text{ctx}} \), because it only relates terms of the same type. It does not make sense to replace \( \langle e \rangle^V \) with \( \langle e \rangle^n \) inside a CBPV program, because the result would not be well-typed.

Reynolds [24] solves a similar problem when comparing direct and continuation semantics of the \( \lambda \)-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 \( \Phi \) from call-by-value computations to call-by-name computations, and \( \Psi \) from call-by-name to call-by-value:

\[ \Gamma \vdash_c M : F \langle \tau \rangle^V \quad \Gamma \vdash_c \Psi, M : \langle \tau \rangle^n \]

Then instead of replacing \( \langle e \rangle^V \) with \( \langle e \rangle^n \) directly, we use \( \Phi \) and \( \Psi \) to convert \( \langle e \rangle^n \) to a computation of the same type as \( \langle e \rangle^V \) (we define this computation formally in Section 6):

\[ \langle \Gamma \rangle^V \rightarrow \langle \Gamma \rangle^n \quad \langle \tau \rangle^n \rightarrow \langle \tau \rangle^V \quad F \langle \tau \rangle^V \]

This behaves like a call-by-name computation, but has the same type as a call-by-value computation. (We could instead have chosen to convert \( \langle e \rangle^V \) into a computation of the same type as \( \langle e \rangle^n \). As we show in Section 6.1, this choice is arbitrary.) 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 22 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 17. A Galois connection consists of two posets \( X, Y \) and two monotone functions \( \phi : X \rightarrow Y, \psi : Y \rightarrow X \), such that \( x \subseteq \psi(\phi x) \) for all \( x \in X \) and \( \phi(\psi y) \subseteq y \) for all \( y \in Y \).

The syntactic maps \( \Phi_\tau \) and \( \Psi_\tau \) are defined in Figure 6. (We use some extra variables in the definition, which are assumed to be fresh.) The maps \( \Phi_\tau \) from call-by-value computations first evaluate the computation, and then map the result to call-by-name using \( \Phi_\tau \), which has the following typing:

\[ \Gamma \vdash_c V : \langle \tau \rangle^V \rightarrow \Gamma \vdash_c \Phi_\tau V : \langle \tau \rangle^n \]

Given any model of CBPV, we correspondingly define morphisms \( \phi_\tau : T[\tau]^V \rightarrow U_T[\tau]^n \) and \( \psi_\tau : U_T[\tau]^n \rightarrow T[\tau]^V \) in Figure 7 (where \( \phi_\tau : [\tau]^V \rightarrow U_T[\tau]^n \)). These morphisms are the interpretations of \( \Phi \) and \( \Psi \) in the following sense.

Lemma 18. Given any model of CBPV, if \( \Gamma \vdash_c M : F \langle \tau \rangle^V \) then \( [\Phi_\tau M] = \phi_\tau \circ [M] \), and if \( \Gamma \vdash_c N : \langle \tau \rangle^n \) then \( [\Psi_\tau N] = \psi_\tau \circ [N] \).

Proof sketch. By induction on the type \( \tau \). Each case is an easy calculation. 

\[ \triangleleft \]
\[ \Phi_\tau M = M \text{ to } x. \hat{\Phi}_\tau x \]
\[ \Phi_{\text{unit}} V = \text{return } V \]
\[ \Phi_{\tau_1 \times \tau_2} V = \text{match } V \text{ with } (z_1, z_2). \lambda \{ (\hat{\Phi}_\tau z_1), 2. (\hat{\Phi}_\tau z_2) \} \]
\[ \Phi_{\text{bool}} V = \text{return } V \]
\[ \Phi_{\tau \rightarrow \tau'} V = \lambda x: U[\tau]^n. \Psi_\tau (\text{force } x) \text{ to } y. (y' \text{ force } V) \text{ to } z. \hat{\Phi}_\tau z \]

**Figure 6** Syntactic maps \( \Phi \) from call-by-value to call-by-name and \( \Psi \) from call-by-name to call-by-value

\[ \Phi_\tau : T[\tau]^n \rightarrow U_T[\tau]^n \]
\[ \phi_\tau = \hat{\phi}_\tau^1 \]

\[ \hat{\phi}_\tau : [\tau]^n \rightarrow U_T[\tau]^n \]
\[ \hat{\phi}_{\text{unit}} = \eta_1 : 1 \rightarrow T1 \]
\[ \hat{\phi}_{\tau_1 \times \tau_2} = (\hat{\phi}_\tau_1 \circ \pi_1, \hat{\phi}_\tau_2 \circ \pi_2) : [\tau_1]^n \times [\tau_2]^n \rightarrow U_T[\tau_1]^n \times U_T[\tau_2]^n \]
\[ \hat{\phi}_{\text{bool}} = \eta_2 : 2 \rightarrow T2 \]
\[ \hat{\phi}_{\tau \rightarrow \tau'} = \Lambda((\phi_\tau \circ ev)^1 \circ ([\tau \rightarrow \tau']^n \times \psi_\tau)) : [\tau]^n \Rightarrow T[\tau']^n \rightarrow U_T[\tau]^n \Rightarrow U_T[\tau']^n \]

\[ \psi_\tau : U_T[\tau]^n \rightarrow T[\tau]^n \]
\[ \psi_{\text{unit}} = id_{T1} : T1 \rightarrow T1 \]
\[ \psi_{\tau_1 \times \tau_2} = \text{seq}_{[\tau_1]^n \times [\tau_2]^n} : U_T[\tau_1]^n \times U_T[\tau_2]^n \rightarrow T([\tau_1]^n \times [\tau_2]^n) \]
\[ \psi_{\text{bool}} = id_{T2} : T2 \rightarrow T2 \]
\[ \psi_{\tau \rightarrow \tau'} = \eta_{[\tau \rightarrow \tau']^n} \circ (\hat{\phi}_\tau \Rightarrow \psi_\tau) : U_T[\tau]^n \Rightarrow U_T[\tau']^n \rightarrow T([\tau]^n \Rightarrow T[\tau']^n) \]

**Figure 7** Semantic morphisms \( \phi \) from call-by-value to call-by-name and \( \psi \) from call-by-name to call-by-value
Galois connecting call-by-value and call-by-name

For the rest of this section, we show that in every model of CBPV that satisfies certain conditions (the assumptions of Theorem 22 below), the functions
\[
\phi_{\tau} \circ - : C(W, T[\tau]^\nu) \to C(W, U_T[\tau]^\nu) \quad \psi_{\tau} \circ - : C(W, U_T[\tau]^\nu) \to C(W, T[\tau]^\nu)
\]
form a Galois connection for every \( \tau \) and \( W \in |C| \). This is the key step in the proof of our reasoning principle (Theorem 25).

First we note that we cannot expect these maps to form Galois connections in general. Consider what happens when we convert a lazy pair \( N = \lambda\{1, N_1, 2, N_2\} \) of type \( \text{unit} \times \text{unit} \) into call-by-value, and then back into call-by-name:

\[
\Phi_{\text{unit} \times \text{unit}}(\Psi_{\text{unit} \times \text{unit}} \{N\}) \equiv \lambda\{1, N_1 \, \text{to} \, z_1, 1, N_2 \, \text{to} \, z_2, \text{return} \, z_1, 2, N_1 \, \text{to} \, z_1, 2, N_2 \, \text{to} \, z_2, \text{return} \, z_2\}
\]

The \( i \)th projection of \( N \) evaluates only \( N_i \), but the \( i \)th projection of \( \Phi_{\text{unit} \times \text{unit}}(\Psi_{\text{unit} \times \text{unit}} \{N\}) \) evaluates both \( N_1 \) and \( N_2 \). We therefore cannot expect \( \Phi_{\text{unit} \times \text{unit}}(\Psi_{\text{unit} \times \text{unit}} \{N\}) \approx^\text{ftr} N \) to hold in general (and semantically, cannot expect \( \phi_{\text{unit} \times \text{unit}} \circ \psi_{\text{unit} \times \text{unit}} \circ [N] \subseteq [N] \) to hold), because moving from the left-hand side to the right-hand side discards side-effects. There is also a problem in the other direction. Converting a strict pair \( M \) of type \( F(\text{unit} \times \text{unit})^\nu = F(\text{unit} \times \text{unit}) \) to call-by-name and back duplicates the side-effects of \( M \):

\[
\Psi_{\text{unit} \times \text{unit}}(\Phi_{\text{unit} \times \text{unit}} M) \equiv M \, \text{to} \, z \, . \, \text{match} \, z \, \text{with} \, (z_1, z_2).
\]
\[
M \, \text{to} \, z' \, . \, \text{match} \, z' \, \text{with} \, (z'_1, z'_2).
\]
\[
\text{return} \, (z_1, z'_2)
\]

Hence converting between the two interpretations of pairs can both discard and duplicate computations (just as call-by-name does).

There is a similar issue for functions. Consider what happens when we convert a CBPV computation \( M : F(\text{bool} \to \text{bool})^\nu = F(U(\text{bool} \to F\text{bool})) \) to call-by-name and then back to call-by-value. By doing this we obtain a computation that immediately returns:

\[
\Psi_{\text{bool} \to \text{bool}}(\Phi_{\text{bool} \to \text{bool}} M) \equiv \text{return thunk} \lambda x : \text{bool}. M \, \text{to} \, z . x \, . \, \text{force} \, z
\]

The computation \( M \) may cause side-effects before producing a (thunk of a) function; but \( \Psi_{\text{bool} \to \text{bool}}(\Phi_{\text{bool} \to \text{bool}} M) \) does not. Thus in general (e.g. if side-effects include mutable state), we cannot expect to have \([M] \subseteq \psi_{\text{bool} \to \text{bool}} \circ \phi_{\text{bool} \to \text{bool}} \circ [M]\), 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 \( M \).

Based on these observations, we add further constraints on models to ensure that we get Galois connections. This is where the restrictions on side-effects appear: the constraints are only satisfied in certain cases (they are for each of our three examples, but not e.g. for any adequate model of mutable state). They are that computations are \textit{lax discardable}, \textit{lax copyable} and \textit{lax thunkable}. The non-lax versions of these properties were first defined by Führmann [6]. Table 1 defines the three properties for computations and for morphisms. Given any adequate model, each property on computations \( \Gamma \vdash M : FA \) holds whenever the corresponding property for the morphism \([M]\) holds. The three constraints on side-effects have a similar role to centrality in Section 4.

To obtain Galois connections at function types below, we will assume that all morphisms \( f : X \to TY \) are thunkable inside the model; this is equivalent to the underlying monad \( T \) being \textit{lax idempotent}. 
Lax discardable  \[ M \text{ to } x. \text{return}() \preceq_{\text{ctx}}^\Gamma \text{ return}() \]

Lax copyable  \[ M \text{ to } x. \text{return}(x,x) \preceq_{\text{ctx}}^\Gamma M \text{ to } x_1. M \text{ to } x_2. \text{return}(x_1,x_2) \]

Lax thunkable  \[ M \text{ to } x. \text{return thunk} \text{ return} x \preceq_{\text{ctx}}^\Gamma \text{ return thunk} M \]

| Table 1 | The three side-effect axioms, defined for computations, morphisms |

Definition 19. A strong Poset-monad \( T \) is lax idempotent\(^3\) when \( T\eta_Y \sqsubseteq \eta_{TY} \) for every object \( Y \in \mathcal{C} \).

This is a strong property. In particular, it implies all of the others:

Lemma 20. In any CBPV model for which \( T \) is lax idempotent, every morphism \( f : X \to TY \) is lax discardable, lax copyable, and lax thunkable.

To obtain Galois connections on product types, it is enough for morphisms to be lax discardable and lax copyable by the following lemma.

Lemma 21. Suppose that every morphism \( f : X \to TY \) is lax discardable and lax copyable. If \( \phi_\tau \circ \psi_\tau_i \sqsubseteq id_U_{\tau_i} \times id_U_{\tau_2} \) and \( id_T[\tau_i]^n \sqsubseteq \psi_\tau \circ \phi_\tau_i \) for each \( i \in \{1,2\} \), then

\[ U_T \phi_{\tau_1 \times \tau_2} \circ \psi_{\tau_1 \times \tau_2} \sqsubseteq id_{U_{\tau_1} \times U_{\tau_2}} \quad id_T[\tau_i]^n \times id_T[\tau_2]^n \sqsubseteq \psi_{\tau_1 \times \tau_2} \circ \phi_{\tau_1 \times \tau_2} \]

To get Galois connections on function types, we require morphisms to be lax thunkable, i.e. that \( T \) is lax idempotent. Since this implies that morphisms are lax discardable and lax copyable, this is also enough for product types. We also trivially have Galois connections for \text{unit} \ and \text{bool}. Hence we arrive at the main theorem of this section:

Theorem 22. Given a CBPV model in which \( T \) is lax idempotent, the functions

\[ \phi_\tau \circ - : C(W,T[\tau]^n) \to C(W,U_T[\tau]^n) \quad \psi_\tau \circ - : C(W,U_T[\tau]^n) \to C(W,T[\tau]^n) \]

form a Galois connection for every source-language type \( \tau \) and object \( X \in \mathcal{C} \).

Proof sketch. By induction on the type \( \tau \). This is trivial for \text{unit} \ and \text{bool}. For product types we use Lemma 21. For function types both of the required inequalities use the fact that \( T \) is lax idempotent.

\[^3\] Lax idempotent Poset-monads are a special case of lax idempotent 2-monads, which are well-known, and are often called Kock-Zöberlein monads [10].
Galois connecting call-by-value and call-by-name

If the model is adequate, it follows from this theorem that the maps $\Phi_{\tau}$ and $\Psi_{\tau}$ on terms form a Galois connection (with respect to $=_\text{ctx}$), by Lemma 18.

**Example 23.** 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 $T\eta_Y = id_Y = \eta_T Y$. For divergence, the monad is lax idempotent because the left hand side of $T\eta_Y t \sqsubseteq \eta_T Y t$ is $\bot$ when $t = \bot$ (intuitively, we can thunk diverging computations), and otherwise the two sides are equal. For nondeterminism, we have $T\eta_Y S = \downarrow\{ y \mid y \in S \} \sqsubseteq \downarrow\{ y \}$ because $\downarrow\{ y \}$ for every $y \in S$ (intuitively, we can postpone nondeterministic choices). Thus we obtain Galois connections in each of these three cases.

6 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 $\Phi$ and $\Psi$ defined above, to form a CBPV computation of the same type as the call-by-value translation:

$$\langle \Gamma \rangle^\nu \longrightarrow \langle \Gamma \rangle^n \xrightarrow{\langle \! e \! \rangle^\nu} \langle \tau \rangle^n \longrightarrow F \langle \tau \rangle^\nu$$

We first define this composition precisely. The arrow on the right is just given by applying $\Psi_{\tau}$. The arrow on the left is a substitution $\hat{\Phi}_{\tau}$ from terms in call-by-name contexts $\{ \Gamma \}^n$ to terms in call-by-value contexts $\{ \Gamma \}^\nu$. Given any source-language typing context $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$, we define $\hat{\Phi}_{\tau} = x_1 \mapsto \text{thunk} (\hat{\Phi}_{\tau_1}(x_1), \ldots, x_n \mapsto \text{thunk} (\hat{\Phi}_{\tau_n}(x_n))$. If $\Gamma \vdash e : \tau$ then $\langle \Gamma \rangle^\nu \vdash_e \Psi_{\tau}(\langle e \rangle^\nu[\hat{\Phi}_{\tau}]) : F \langle \tau \rangle^\nu$, which has the same typing as $\langle e \rangle^\nu$. The statement we wish to prove is $\langle e \rangle^\nu \sqsubseteq_{\text{ctx}}^{\nu} \Psi_{\tau}(\langle e \rangle^\nu[\hat{\Phi}_{\tau}])$. Again we reason using the denotational semantics. Given a CBPV model, the interpretation of the substitution $\hat{\Phi}_{\tau}$ is a morphism $\hat{\phi}_{\tau} : [\Gamma]^\nu \to [\Gamma]^n$, given by $\hat{\phi}_c = id_1$ and $\hat{\phi}_{\tau_1 \times \tau_2} = \hat{\phi}_\tau \times \hat{\phi}_\tau$. If the model is adequate, to show our goal $\langle e \rangle^\nu \sqsubseteq_{\text{ctx}}^{\nu} \Psi_{\tau}(\langle e \rangle^\nu[\hat{\Phi}_{\tau}])$, it suffices to show that $\langle e \rangle^\nu \sqsubseteq \psi_{\tau} \circ \langle e \rangle^\nu \circ \hat{\phi}_{\tau}$.

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

**Lemma 24.** In every CBPV model for which the functions

$$\phi_{\tau} \circ - : C(W, T[\tau]^\nu) \to C(W, U[\tau]^n) \quad \psi_{\tau} \circ - : C(W, U[\tau]^n) \to C(W, T[\tau]^\nu)$$

form Galois connections for all $\tau$, $W$, we have $\langle e \rangle^\nu \sqsubseteq \psi_{\tau} \circ \langle e \rangle^\nu \circ \hat{\phi}_{\tau}$ 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 involve use properties of Galois connections. The equalities follow from properties of products, exponentials, and $T$-algebras.

$$\langle e e' \rangle^\nu = (\langle ev \rangle \circ (\pi_2, \langle e' \rangle^\nu \circ \pi_1))^\dagger \circ \langle id, \langle e \rangle^\nu \rangle$$

$$\sqsubseteq \psi_{\tau'} \circ \phi_{\tau'} \circ (\langle ev \rangle \circ (\pi_2, \psi_{\tau} \circ \langle e' \rangle^\nu \circ \phi_{\tau} \circ \pi_1))^\dagger \circ \langle id, \psi_{\tau \to \tau'} \circ \langle e \rangle^\nu \circ \hat{\phi}_{\tau} \rangle$$

$$\quad \sqsubseteq \psi_{\tau'} \circ \lambda^{-1} (\phi_{\tau \to \tau'} \circ \psi_{\tau \to \tau'} \circ \langle e' \rangle^\nu \circ \langle id, \langle e' \rangle^\nu \rangle \circ \hat{\phi}_{\tau}$$

$$\quad \sqsubseteq \psi_{\tau'} \circ \langle e' \rangle^\nu \circ \hat{\phi}_{\tau}$$
Theorem 22 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 25. Recall that a program relation $\preceq$ is a family of relations on CBPV programs, and that each program relation induces a contextual preorder $\preceq_{\text{ctx}}$. Given any program relation $\preceq$, to show that the call-by-value and call-by-name translations of source-language expressions are related by $\preceq_{\text{ctx}}$ it is enough to find an adequate model involving a lax idempotent $\mathcal{T}$.

**Theorem 25 (Relationship between call-by-value and call-by-name).** Suppose we are given a program relation $\preceq$, and a model of CBPV that is adequate with respect to $\preceq$, and has a lax idempotent $\mathcal{T}$. If $\Gamma \vdash e : \tau$ then

$$\langle e \rangle^\tau \preceq_{\text{ctx}} \Psi_r (\langle e \rangle^n [\hat{\Phi}_r])$$

**Proof.** Theorem 22 implies that $\phi$ and $\psi$ form Galois connections, and then Lemma 24 implies $\langle e \rangle^\tau \subseteq \psi_r \circ [\hat{\tau}]^n \circ \hat{\Phi}_r$. The result follows from adequacy of the model and Lemma 18. ▶

The generality of this theorem comes from two sources. First, we consider arbitrary program relations $\preceq$. 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 `unit` or `bool`). This corollary is closer to the standard results that are proved for specific side-effects.

**Corollary 26.** If the assumptions of Theorem 25 hold, then for every closed expression $e$ of type $\tau$, where $\tau \in \{\text{unit, bool}\}$, we have $\langle e \rangle^\tau \preceq \langle e \rangle^n$.

**Proof.** We have $\langle e \rangle^\tau \subseteq \tau_r \circ [\hat{\tau}]^n \circ \hat{\Phi}_r$ because both $\psi_r$ and $\hat{\Phi}_r$ are identities. Adequacy implies $\langle e \rangle^\tau \preceq_{\text{ctx}} \langle e \rangle^n$, and hence $\langle e \rangle^\tau \preceq \langle e \rangle^n$.

Our reasoning principle also has a partial converse:

**Lemma 27.** If $\langle e \rangle^\tau \subseteq \psi_r \circ [\hat{\tau}]^n \circ \hat{\Phi}_r$ for each $\Gamma \vdash e : \tau$, then for $X \in \{1, 2\}$ we have $T\eta_X \subseteq \eta_T X$, and every morphism $Y \to TX$ is lax thunkable.

**Proof sketch.** The first step is to show that $id \subseteq \psi_r \circ \hat{\Phi}_r$ for every $\tau$, by applying the assumption to the expression $x : \text{unit} \to \tau \vdash x () : \tau$. It follows for each $\tau' \in \{\text{unit, bool}\}$ that $id_{\tau'(T\eta_X \subseteq \eta_T X)} \subseteq \psi_{\text{unit} \to \tau'} \circ \hat{\Phi}_{\text{unit} \to \tau'} = \eta_{1 \to T\eta_X} \circ id_{1 \to T\eta_X}$, and hence

$$T\eta_{\tau' \cdot \tau} \circ id_{T\eta_{\tau' \cdot \tau}} \circ T\eta_{\tau' \cdot \tau} \circ id_{T\eta_{\tau' \cdot \tau}} \circ T\eta_{\tau' \cdot \tau} = \eta_{T\eta_{\tau' \cdot \tau}}$$

as required. ▶

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 $\mathcal{T}$ is lax idempotent. After extending the inductive proof of Lemma 24 with cases for the extra syntax, we can apply our relationship between call-by-value and call-by-name (Theorem 25).

In particular, we can apply Corollary 26 to relate source-language programs. For no side-effects, this shows for each $e : \text{bool}$ that there is some $V$ such that $\langle e \rangle^\tau \downarrow \text{return} V$ and $\langle e \rangle^n \downarrow \text{return} V$. 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 $\langle e \rangle^\tau \downarrow \text{return} V$ implies $\langle e \rangle^n \downarrow \text{return} V$ 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.
6.1 Equivalent reasoning principles

Our reasoning principle (Theorem 25) shows that a call-by-value computation can be replaced with a call-by-name computation when the call-by-name computation is composed with our maps between evaluation orders. Here we have made an arbitrary choice: by composing the call-by-value computation with the maps instead, there are three other reasoning principles we can derive.

We use the properties of Galois connections to derive the three alternative reasoning principles from the one above. The first alternative, which converts the result of the call-by-value computation to call-by-name, is almost immediate: \( \Phi_\tau \langle e \rangle^v \preceq_{ctx} \langle e \rangle^n [\hat{\Phi}_\Gamma] \) because

\[
\phi_\tau \circ \langle e \rangle^v \subseteq \phi_\tau \circ \psi_\tau \circ \langle e \rangle^n \circ \hat{\phi}_\Gamma \subseteq \langle e \rangle^n \circ \hat{\phi}_T
\]

where the first inequality is the reasoning principle we have already given, and the second is a property of Galois connections. The other two involve converting call-by-name contexts into call-by-value contexts. These are more difficult because call-by-name contexts contain unevaluated computations, but call-by-value contexts contain results of computations. Hence to convert a call-by-name context to a call-by-value context we must force the evaluation of computations.

To derive these last two reasoning principles, we first define a morphism \( \psi_T : [\Gamma]^n \to T[\Gamma]^y \) for each source-language context \( \Gamma \):

\[
\psi_0 = \eta_1 : 1 \to T1 \quad \psi_{T,x,T} = seq \circ (\psi_T \times \psi_T) : [\Gamma]^n \times U_T [\tau]^n \to T([\Gamma]^y \times [\tau]^y)
\]

These morphisms map call-by-name contexts to call-by-value contexts in the denotational semantics.

The four reasoning principles are summarised in Figure 8. We give each of them syntactically, in the form of an instance of \( \preceq_{ctx} \), and semantically as an instance of \( \subseteq \). Each square in the figure is the interpretation of the instance of \( \preceq_{ctx} \) above it, so adequacy enables us to derive the reasoning principles on the syntax by showing these instances of \( \subseteq \). The reasoning principle on the top left is the one we derived as Theorem 25. The top right is Equation (1). The bottom two are the more complex ones that involve converting call-by-name contexts into call-by-value contexts.

The bottom two reasoning principles can be derived from the top left if \( T \) is lax idempotent. This can be shown by applying the following lemma to \( f = \langle e \rangle^v \) and \( g = \langle e \rangle^n \).

\textbf{Lemma 28.} If \( T \) is lax idempotent, then for each \( f : [\Gamma]^y \to T[\tau]^y \) and \( g : [\Gamma]^n \to U_T [\tau]^n \), the following four inequalities are equivalent:

\[
f \subseteq \psi_\tau \circ g \circ \hat{\phi}_T \quad \phi_\tau \circ f \subseteq g \circ \hat{\phi}_T \quad \phi_\tau \circ f^1 \circ \psi_T \subseteq g \quad f^1 \circ \psi_T \subseteq \psi_\tau \circ g
\]

7 Related work

Comparing evaluation orders  Plotkin [22] and many others (e.g. [8]) 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 [16, 17, 7]. We suspect our technique could also be adapted to these. It might also be possible to recast some of our work in terms of the duality between call-by-value and call-by-name [4, 2, 27].
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 [24] to relate direct and continuation semantics of the lambda calculus, and later used by others (e.g. [19, 11, 1, 5]). 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 [25, 12, 26] 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.

8 Conclusions

In this paper, we give a general reasoning principle (Theorem 25) 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 26). 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
Galois connecting call-by-value and call-by-name

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


