

# THEORETICAL PEARL

## *Call-by-value and call-by-name: A simple proof of a classic theorem*

DARIUSZ BIERNACKI 

University of Wrocław, Wrocław, Poland  
(e-mail: [dabi@cs.uni.wroc.pl](mailto:dabi@cs.uni.wroc.pl))

JAMES MCKINNA 

Heriot-Watt University, Edinburgh EH14 4AS, UK  
(e-mail: [j.mckinna@hw.ac.uk](mailto:j.mckinna@hw.ac.uk))

FILIP SIECZKOWSKI 

Heriot-Watt University, Edinburgh EH14 4AS, UK and University of Wrocław, Wrocław, Poland  
(e-mail: [efes@cs.uni.wroc.pl](mailto:efes@cs.uni.wroc.pl))

---

### Abstract

One of the natural problems of operational semantics is to characterise the relationship between eager and lazy evaluation. In the context of  $\lambda$ -calculus, this is expressed by the classic theorem that call-by-value evaluation of a program to (weak-head) normal form can always be simulated by a call-by-name evaluation. While the statement and intuition behind it are simple and clear, naive attempts at proof famously fail: the result is usually established as a consequence of the more complex standardisation theorem. In this work, we develop and formalise a novel and lightweight inductive approach to tackle the problem of simulation between two semantics for a single calculus, but with different evaluation orders. We exercise our method on the classic call-by-value and call-by-name example and report on methodological takeaways suggested by our approach, in particular what effect the flavour of semantics chosen has on the proof.

---

### 1 Introduction

Untyped  $\lambda$ -calculus is one of the prototypical models of computation since the dawn of computer science (Barendregt, 1985): it remains the foundation of functional programming languages and a common model language introduced in formal studies of semantics (Landin, 1964; Reynolds, 1972; Plotkin, 1975; Reynolds, 1998; Pierce, 2002; Wadler *et al.*, 2022). While originally introduced as a calculus, with reductions performed in arbitrary contexts, in modern applications to modelling programming languages, it usually appears with (operational) semantics given only for *closed* terms, which model complete *programs*. In such a setting, the main semantic distinction is whether function

application is *eager*, i.e., evaluates the argument before passing it to the function, giving a call-by-value, or *applicative*, strategy or *lazy*,<sup>1</sup> i.e., passes the argument to the function without evaluating it, giving rise to the call-by-name, or *normal*, strategy (Plotkin, 1975). It is trivial to observe that sometimes the call-by-name strategy terminates while call-by-value does not. A natural, and much more difficult question is whether the converse holds: if a program terminates under eager evaluation, does it also terminate if evaluated lazily? While the answer to this is well known, the naive approach at a proof fails rather spectacularly, as detailed below. The folklore proofs hinge on rather sophisticated results in the reduction theory of the  $\lambda$ -calculus, such as the left-to-right standardisation theorem (Curry & Feys, 1958) that for the  $\lambda$ -calculus is typically shown as a corollary of a theorem known as semi-standardisation or factorisation of  $\beta$ -reduction through head reduction (Mitschke, 1979), or, even more directly, McKinna and Pollack’s factorisation of  $\beta$ -reduction through weak-head reduction (McKinna & Pollack, 1999), which is an inductive formalisation of Plotkin’s definition (Plotkin, 1975), and which also implies left-to-right standardisation.

In this paper, we present a simple method of building a simulation proof that relies only on the two operational semantics and inductively defined predicates, rather than on a full-fledged reduction theory and highly non-trivial general theorems characterizing this reduction theory. First, such a reduction theory may not even be available or interesting to be developed, and when it is, each modification of the calculus or programming language under consideration requires the corresponding factorisation and/or standardisation theorems to be re-established. In contrast, our method does not require any external results. Second, we believe that our approach sheds far more light on the relationship between the two semantics than the folklore proofs, by providing an explicit relation between the two worlds. Finally and equally importantly, we find our method both scalable and simple enough to warrant its inclusion in introductory courses on semantics.

*Contributions.* Concretely, we make the following contributions:

- We establish a novel, lightweight methodology for proving simulation results between multiple semantics of a language, which differ in terms of order of evaluation.
- We formalise our approach using the prototypical simulation between call-by-value and call-by-name  $\lambda$ -calculus, in a range of semantic flavours.
- We mechanise the results using Agda, highlighting the simplicity, scalability and robustness of our proof method.

## 2 Proving the simulation

To set the stage, we recall the syntax of untyped lambda-calculus, our language of choice, in Figure 1. Note that the presentation is scope-aware: the set  $\Lambda^X$  is the set of lambda-terms with free variables drawn from the set  $X$ . While the presentation is a variant of Bird and Paterson’s nested-type representation (1999) and is fully compatible with the functorial approach to binding (Fiore et al., 1999), this is not fundamental: one can just as easily use a standard definition of lambda terms and take  $\Lambda^X = \{e \in \Lambda \mid \text{FV}(e) \subseteq X\}$ .

<sup>1</sup> In this paper we are lax about “laziness” in not considering call-by-need evaluation (Wadsworth, 1971).

$$\Lambda^X \ni e^X ::= x_{(x \in X)} \mid \lambda x. e^{X \uplus \{x\}} \mid e_1^X e_2^X$$

Fig. 1. The (scope-aware) syntax of untyped  $\lambda$ -calculus. We elide the sets of free variables except where their inclusion aids clarity. We take closed terms, i.e.,  $p \in \Lambda^\emptyset$ , as complete *programs*.

With the syntax of the calculus defined, we can define the call-by-value and call-by-name semantics. However, before we do that, we discuss the flavours of semantics and their impact on the difficulty of the simulation proof.

**The problem with the naive approach and the impact of the semantic framework.** The intuition behind why the simulation theorem holds is very simple and convincing: since we reduce the same program, at any step in the call-by-name evaluation, it will work on a program that was evaluated by the call-by-value version at some point (possibly significantly earlier). However, this intuition is difficult to reconcile with the usual presentation of semantics, where reduction of  $\beta$ -redexes is performed via substitution (Plotkin, 1975). Immediately after the first application with a non-value argument is encountered, and the argument in the function position evaluated to matching values, we substitute *different* terms on the call-by-value and call-by-name sides. Thus, we are no longer reducing the same program on both sides and the simple-minded inductive argument breaks down.

This breakdown of our proof attempt is deeply unsatisfactory: of course, the programs are slightly different, but intuition tells us that they still have the same *structure*, and that wherever that structure diverges, it does so in a ‘safe’ way. However, it is far from obvious how to capture this intuition formally. In fact, it turns out that it is easiest by utilising a flavour of semantics where at any point the *code* that is evaluated is a sub-expression of the original program – a variant of a *subterm property* (Accattoli & Dal Lago, 2012). If we set up our semantics in this way, the original intuition that call-by-name only ever evaluates pieces of code that call-by-value evaluated somewhere along the way can be made precise in a natural fashion.

The simplest semantic format with this property is the *environment* semantics (Kahn, 1987), which is a big-step semantics with explicit environments, closely related to the usual way of defining evaluators using closures and environments rather than a meta-level substitution operation (Reynolds, 1972). One of its useful features, from an implementation standpoint, is that at any point the expression that is evaluated can be represented as a ‘pointer’ into the program (i.e., the program text does not change under evaluation) – and since this expression can be open, it is the environment that provides the interpretation for the free variables. In our application, we can use this fact to drive the simulation argument, since we can assert that the two semantics always evaluate the *same* expression, albeit in somewhat different, but crucially, *appropriately related*, environments.

## 2.1 The environment semantics for call-by-value and call-by-name

In contrast to the classic substitution semantics, before we define the evaluation judgements, we need to construct several pieces of instrumentation. For the sake of symmetry, we define those in matching ways: for each semantics, the three components comprise

$$\begin{array}{ll}
\text{Clo}_v \ni c ::= \langle e^X, \rho^X \rangle & \text{Clo}_n \ni c ::= \langle e^X, \sigma^X \rangle \\
\text{Val}_v \ni v ::= \langle \lambda x. e^{X \uplus \{x\}}, \rho^X \rangle & \text{Val}_n \ni v ::= \langle \lambda x. e^{X \uplus \{x\}}, \sigma^X \rangle \\
\rho^X \in \text{Env}_v^X = X \rightarrow \boxed{\text{Val}_v} & \sigma^X \in \text{Env}_n^X = X \rightarrow \boxed{\text{Clo}_n} \\
\\
(\text{VAR}) \frac{x \in X}{\langle x, \rho^X \rangle \Downarrow_v \rho(x)} & (\text{VAR}) \frac{x \in X \quad \boxed{\sigma(x) \Downarrow_n v}}{\langle x, \sigma^X \rangle \Downarrow_n v} \\
\\
(\text{LAM}) \frac{}{\langle \lambda x. e, \rho \rangle \Downarrow_v \langle \lambda x. e, \rho \rangle} & (\text{LAM}) \frac{}{\langle \lambda x. e, \sigma \rangle \Downarrow_n \langle \lambda x. e, \sigma \rangle} \\
\\
(\text{APP}) \frac{\langle e_1, \rho \rangle \Downarrow_v \langle \lambda x. e, \rho' \rangle \quad \boxed{\langle e_2, \rho \rangle \Downarrow_v v}}{\langle e, \rho'[x \mapsto v] \rangle \Downarrow_v v'} & (\text{APP}) \frac{\langle e_1, \sigma \rangle \Downarrow_n \langle \lambda x. e, \sigma' \rangle \quad \boxed{\langle e, \sigma'[x \mapsto \langle e_2, \sigma \rangle] \rangle \Downarrow_n v'}}{\langle e_1 e_2, \sigma \rangle \Downarrow_n v'}
\end{array}$$

Fig. 2. Environment semantics for call-by-value (left) and call-by-name (right). The difference in structure of the two semantics is highlighted by framing the distinctive parts of the two definitions. Note that in both cases, closures and values are closed by construction. To avoid clutter, we elide the sets of free variables except where useful as a clarification.

the types of *closures*, *values*, and *environments*, with the evaluation judgements relating closures to values in their respective semantics.

Starting with the call-by-value semantics, presented in Figure 2 (left), the only kind of values we have are functions, represented by lambda-abstractions with  $X$  free variables, together with a closing environment,  $\rho \in \text{Env}_v^X$ , which are extended once an argument for the function is provided. The environments, defined mutually inductively with values, simply map their domains into values themselves. Finally, closures, which represent complete running programs, are formed from an open expression together with a closing environment. Note that both values and closures implicitly sum over all (finite) sets of free variables. The evaluation judgement matches the usual definition of an evaluator: for the variable case, we simply consult the environment, and encountering a lambda-abstraction finishes the evaluation and packs the function with the environment to yield a value. For the application case, we evaluate both sub-expressions in the environment with which we started. The result of evaluation of the function position yields a lambda-abstraction with some closing environment  $\rho'$ , which can now be extended with the result of the evaluation of the argument to allow us to evaluate the body of the function.

The call-by-name semantics, presented in Figure 2 (right), has a very similar structure. In this case, since the objects stored in the environment are *not* yet evaluated, it is the closures and environments that are mutually inductively defined: closures consist of an open expression and a closing environment, while environments map their domains into closures themselves. Finally, values package a lambda-abstraction with a closing environment, similarly to the call-by-value case. The evaluation judgement again mirrors a standard evaluator: in the variable case we look up the matching *closure* in the environment and continue the computation, and the lambda-abstraction packages the function and the environment as a value, as before. In the case of application, we only evaluate the argument in the function position to obtain a function with a new environment, and continue

$$\begin{array}{c}
\frac{\rho \sim^e \sigma}{\langle e, \rho \rangle \sim^c \langle e, \sigma \rangle} \qquad \frac{\rho \sim^e \sigma}{\langle \lambda x. e, \rho \rangle \sim^v \langle \lambda x. e, \sigma \rangle} \qquad \frac{v \sim^v u \quad c \Downarrow_n u}{v \sim c} \\
\rho^X \sim^e \sigma^X \iff \forall x \in X. \rho(x) \sim \sigma(x)
\end{array}$$

Fig. 3. The simulation relation for environment semantics.

the evaluation of the function's body in that environment extended with the argument program paired with its initial matching environment. Note that in both cases the well-scoped representation of terms and environments helps ensure that all the environments are used correctly.

## 2.2 The simulation relation

Having defined the semantics, we can reify our initial intuition that a running call-by-name program simulates a call-by-value one if they compute the same program — although we also need to account for the fact that the environments cannot be arbitrary. In total, we define four mutually inductive relations (some of which could be inlined). These comprise similarity relations on closures, values, environments (relative to the same set of variables) and *function arguments*. This last case relates arguments in call-by-value, i.e., values, to arguments in call-by-name, i.e., closures. Figure 3 presents the complete definition.

The relations on closures,  $\sim^c$ , and values,  $\sim^v$ , are defined according to our intuition: the expression is identical on both sides, while the closing environments need to be related. This is the essential use of the subterm property — the distinction between the source expressions and their closing environments allows us to insist that the former are *identical*, while the latter need merely to be related. This also suggests that our approach would scale naturally if, for instance, we had additional kinds of values. The relation on environments, written  $\sim^e$ , is a simple point-wise extension of the relation on arguments,  $\sim$ , which matches call-by-value values to call-by-name closures. This is achieved by asserting that the call-by-name closure evaluates to a value that is related to its call-by-value counterpart. Note that the relation on environments holds trivially for the pair of *empty* environments, and thus that any closures constructed from a program (i.e., a closed expression), paired with the corresponding empty environment, are related.

## 2.3 The proof

With the simulation relations defined, we can finally state the main inductive lemma, as the usual simulation square.

**Lemma 2.1.** *If two closures are related,  $c_v \sim^c c_n$ , and the call-by-value closure evaluates to a value,  $c_v \Downarrow_v v$ , then the call-by-name closure also evaluates to some value  $w$ ,  $c_n \Downarrow_n w$ , which is, moreover, related to its counterpart,  $v \sim^v w$ .*

Note that in light of our definition of simulation relation, the conclusion of this lemma can also be stated simply as an instance of our fundamental relation on arguments,  $v \sim c_n$ .

**Proof** The proof proceeds by induction on the structure of the evaluation judgement. We have three cases, which determine the structure of  $c_v$ :

1. (VAR) In this case, both  $c_v$  and  $c_n$  are the same variable,  $x \in X$ , in related environments  $\rho \sim^e \sigma$ , both ranging over  $X$ . Since the environments are related, the resulting value,  $\rho(x)$  is related to the closure  $\sigma(x)$  as arguments – that is, we have  $\sigma(x) \Downarrow_n w$  and  $\rho(x) \sim^v w$ . However, by the call-by-name version of (VAR), we have  $\langle x, \sigma \rangle \Downarrow_n w$ , which ends this case of the proof.
2. (LAM) In this case, both sides terminate with a value: the lambda-abstraction encountered is the same, and the environments are related by the similarity assumption. Thus, the corresponding values are also related.
3. (APP) Finally, in the application case, both closures are formed as the application of the same two expressions, say,  $e_1 e_2$  with related environments  $\rho \sim^e \sigma$ . From the evaluation assumption for  $e_1$ , which states that  $\langle e_1, \rho \rangle \Downarrow_v \langle \lambda x.e, \rho' \rangle$ , we learn by induction hypothesis that the evaluation by name gives us the *same* function in a related environment, i.e., that  $\langle e_1, \sigma \rangle \Downarrow_n \langle \lambda x.e, \sigma' \rangle$  such that  $\rho' \sim^e \sigma'$ . The evaluation assumption for  $e_2$  gives us  $\langle e_2, \rho \rangle \Downarrow_v v$  which, by induction hypothesis, ensures that  $v \sim \langle e_2, \sigma \rangle$ . This allows us to extend the environments  $\rho'$  and  $\sigma'$  with  $v$  and  $\langle e_2, \sigma \rangle$  respectively, getting  $\rho'[x \mapsto v] \sim^e \sigma'[x \mapsto \langle e_2, \sigma \rangle]$ . The proof now follows by the induction hypothesis for the evaluation of  $e$ , using the (APP) rule to ‘extend’ the evaluation on the call-by-name side. ■

The lemma now serves to establish our original theorem as an immediate corollary.

**Corollary 2.2.** *If a closed program, evaluated in an empty environment, terminates in call-by-value, then it also terminates in call-by-name.*

While the simulation proof presented above is elegant and simple, it raises a pertinent question: does the approach we followed here scale to other calculi and other semantic formats, or was it a happy accident that we were able to construct the similarity relations in this particular case. In the following section, we give evidence that the approach is in fact rather robust.

### 3 Substitutions and small steps: adaptations to other semantic formats

In this section, we sketch a way to adapt our proof to other common semantic formats.<sup>2</sup> First, let us consider a substitution-based semantics (Reynolds, 1998, Chapter 10). It is clear that we could port the definition in Figure 3 to a substitution-based semantics by existentially quantifying over related substitutions and actually performing them to obtain related programs. This, however, is unwieldy and rather unsatisfactory – and it is sensible to ask whether there is a more natural approach to defining the simulation that would work for substitution-based semantics.

<sup>2</sup> For detailed definitions and other variations, please consult the Agda sources.

$$\begin{array}{c}
\text{BETA} \frac{}{(\lambda x.e) v \mapsto_v e[v/x]} \qquad \text{BETA} \frac{}{(\lambda x.e) e' \mapsto_n e[e'/x]} \\
\\
\text{APPL} \frac{e_1 \mapsto_v e'_1}{e_1 e_2 \mapsto_v e'_1 e_2} \quad \text{APPR} \frac{e \mapsto_v e'}{v e \mapsto_v v e'} \quad \text{APPL} \frac{e_1 \mapsto_n e'_1}{e_1 e_2 \mapsto_n e'_1 e_2}
\end{array}$$

Fig. 4. Small-step, substitution-based semantics: call-by-value (left) and call-by-name (right). The notation  $e[e'/x]$  denotes a substitution of expression  $e'$  for the variable  $x$  in  $e$ ; we take values  $v$  to range over closed lambda abstractions, and only reduce *closed* programs.

It turns out that such an adaptation is possible. We can retain the three relations on values (which now become simply closed lambda abstractions), arguments and expressions – however, the latter relation now considers *open* terms, and is formed by closing the relation for arguments under all the term formers. At the same time, the relation on arguments simply requires that the by-name side evaluates to a value and that the two values are in the relation on values, which itself is just the relation on expressions, specialized to closed lambda abstractions.

Intuitively, this definition expresses that the two programs share the structure until such point (or points) where a substitution occurs: there, the results of substitution need to be related as arguments, forcing the fact that evaluation on the by-name side terminates. It is easy to show that this definition is closed under related substitutions, akin to Lemma 3.1 discussed in the following, thus leading to a fundamental simulation lemma, analogous to the one we proved in the previous section.

While such a construction can account for substitution-based semantics, it remains tied to big-step semantics. This is somewhat limiting: small-step semantics (Plotkin, 1981) are extremely popular (Pierce, 2002; Harper, 2016), both due to conciseness of presentation and ease of use (including, for instance, being able to express non-terminating computations). However, our definition of simulation – in particular, the relation on arguments – seems to rely rather fundamentally on a program being evaluated all the way down to a value. Can we reconcile this notion with a small-step notion of reduction, such as the one presented in Figure 4?

It turns out that the answer is, again, yes. In fact, the notion of relation on arguments can be split in two: a relation on programs that expresses that the lazy program can ‘catch up with’ the eager one, i.e., reduce (in any number of steps) to a program that is structurally<sup>3</sup> related to the eager one. The relation on arguments is then just a special case where the eager side of the relation is required to be a value. Since small-step closure calculi usually entail additional semantic artifacts, we present a version of the simulation for small-step, substitution-based semantics in Figure 5. Note that, in addition to the relations on arguments, and (open) expressions, we have the ‘chase’ relation ( $\rightsquigarrow$ ) — but also, another relation, on *closed* programs. This last relation, written  $\sim^p$ , relates expressions that are related structurally, but with possible discrepancies in the argument position, where the call-by-value side may have already started the process of evaluating the argument. Note that under a lambda, the relation defaults to the usual, structural relation (with the

<sup>3</sup> As before, up to occurrences of related argument substitution.

$$\begin{array}{c}
\frac{}{x \sim^t x} \qquad \frac{e_v \sim^t e_n}{\lambda x. e_v \sim^t \lambda x. e_n} \qquad \frac{e_v \sim^t e_n \quad a_v \sim^t a_n}{e_v a_v \sim^t e_n a_n} \qquad \frac{v \sim e}{v \sim^t e} \\
\\
\frac{p_v \sim^p p'_n \quad p_n \mapsto_n^* p'_n}{p_v \leftarrow p_n} \qquad \frac{v \leftarrow e}{v \sim e} \qquad \frac{e_v \sim^t e_n}{\lambda x. e_v \sim^p \lambda x. e_n} \qquad \frac{p_v \sim^p p_n \quad a_v \leftarrow a_n}{p_v a_v \sim^p p_n a_n}
\end{array}$$

Fig. 5. Simulation definition for small-step, substitution-based semantics. Here,  $\sim^t$  is the open, structural closure of the relation for arguments,  $\sim$ . The later is expressed in terms of the ‘chase’ relation,  $\leftarrow$  which in turn utilises a structural relation on partially evaluated programs,  $\sim^p$ .

appropriately embedded related arguments). This setup leads, via a substitution lemma, to a more relaxed version of the main lemma:

**Lemma 3.1.** *For any related terms over a set  $X \uplus \{x\}$ ,  $e_v \sim^t e_n$  and any related arguments,  $v \sim p$ , the results of substituting the argument for  $x$  on their respective sides are also related,  $e_v[v/x] \sim^t e_n[p/x]$ .*

**Lemma 3.2.** *The chase relation is closed under by-value reduction on the left, i.e., for any chase-related closed programs,  $p_v \leftarrow p_n$ , if the call-by-value side reduces to some program  $p$ ,  $p_v \mapsto_v^* p$ , then the resulting program is still chase-related to  $p_n$ , i.e.,  $p \leftarrow p_n$ .*

The former lemma proceeds by induction on the structure of the similarity of terms, using the assumption whenever variable  $x$  is encountered, while the latter is proved by induction on the reduction sequence, and then by mutual induction on the structure of relations  $\leftarrow$  and  $\sim^p$  (for a single step on the by-value side). Since the chase relation is reflexive, the original theorem remains a simple corollary in the case where  $p$  happens to be a value.

#### 4 Formalisation

In this section, we briefly discuss our Agda (Norell, 2007) formalisation of the results described in previous sections. In particular, we reprise the development of Section 2, based on big-step, environment-based semantics. It is simple enough to allow us to present in its entirety, highlighting how the maturity and sophistication of modern theorem-proving/programming environments can aid in streamlining developments of ‘classical’ metatheoretic results. However, in the associated formalisation, we also work out the development for other styles of semantics, providing adaptations to substitution-based semantics and small-step calculi.

The syntax and operational semantics of our calculi are presented in Figure 6. Since we chose a nested-type representation of well-scoped lambda-terms, à la Bird & Paterson (1999), the presentation is very similar to that of Section 2. The flipside is that the sets of closures and values inhabit  $\text{Set}_1$ , due to quantification over a set of free variables; this is largely inconsequential, and any other well-scoped representation would perform equally well.

```

data Tm (X : Set) : Set where
  ' _ : (x : X) → Tm X
  λ _ : (e : Tm (Ext X)) → Tm X
  · _ : (e1 e2 : Tm X) → Tm X
Prog = Tm ⊥

data Clov : Set1
data Valv : Set1
Envv : Set → Set1

data Valv where
  [λ _ , _] : (e : Tm (Ext X)) (ρ : Envv X) → Valv
Envv X = X → Valv

data Clov where
  [ _ , _ ] : (e : Tm X) (ρ : Envv X) → Clov

data ↓v _ : Clov → Valv → Set1 where
  var : ∀ x (ρ : Envv X) →
    [ ' x , ρ ] ↓v ρ x
  lam : ∀ e (ρ : Envv X) →
    [ λ e , ρ ] ↓v [ λ e , ρ ]
  app : [ e1 , ρ ] ↓v [ λ e , ρ' ] →
    [ e2 , ρ ] ↓v u →
    [ e , ρ' ▷ u ] ↓v v →
    [ e1 · e2 , ρ ] ↓v v

ρ0 : Envv ⊥
ρ0 ()
clov : Prog → Clov
clov p = [ p , ρ0 ]

data Clon : Set1
data Valn : Set1
Envn : Set → Set1

data Valn where
  [λ _ , _] : (e : Tm (Ext X)) (σ : Envn X) → Valn
Envn X = X → Clon

data Clon where
  [ _ , _ ] : (e : Tm X) (σ : Envn X) → Clon

data ↓n _ : Clon → Valn → Set1 where
  var : σ x ↓n w →
    [ ' x , σ ] ↓n w
  lam : ∀ e (σ : Envn X) →
    [ λ e , σ ] ↓n [ λ e , σ ]
  app : [ e1 , σ ] ↓n [ λ e , σ' ] →
    [ e , σ' ▷ [ e2 , σ ] ] ↓n w →
    [ e1 · e2 , σ ] ↓n w

σ0 : Envn ⊥
σ0 ()
clon : Prog → Clon
clon p = [ p , σ0 ]

```

Fig. 6. Agda formalisation of the syntax and operational semantics for call-by-value (left) and call-by-name (right) lambda-calculus.

Note that Agda's support for mutual induction and induction-recursion makes the definitions simple and largely devoid of clutter. We also make extensive use of the abbreviation mechanisms usually called 'implicit syntax' (Pollack, 1990), i.e., arguments which in a given context can be inferred by the typechecker may be omitted from the explicit parametrisation of, e.g., term or datatype structure. Moreover, Agda extends such idiomatic usage with a sophisticated generalisation mechanism that allows us to mimic the standard practice of implicitly universally quantifying free variables in definitions in a formally sound manner.

These convenient features also scale to the case of the definition of the simulation relation, presented in Figure 7, which again mirrors that of Section 2. Note that, even though the relations on values and computations only have one constructor, this is due to simplicity of our calculus: additional value- and computation-formers would be encountered in larger languages (or, indeed, other semantic formats). On the other hand, the simulation on arguments,  $\sim$ , is by its design a single-constructor type, and hence the datatype could be replaced with a record. However, we found that this largely did not affect the proofs or their readability.

```

data  $\sim^v$  : Valv → Clon → Set1
data  $\sim^v$  : Valv → Valn → Set1
 $\sim^e$  : Envv X → Envn X → Set1
data  $\sim^c$  : Clov → Clon → Set1

data  $\sim$  where
   $\sim^v$  : {w : Valn} → v  $\sim^v$  w → cn  $\sim^v$  w → v  $\sim$  cn

data  $\sim^v$  where
  [λ_,_] : (e : Tm (Ext X)) → ρ  $\sim^e$  σ → [λ e, ρ]  $\sim^v$  [λ e, σ]

ρv  $\sim^e$  ρn = ∀ x → ρv x  $\sim$  ρn x

data  $\sim^c$  where
  [_,_] : (e : Tm X) → ρ  $\sim^e$  σ → [e, ρ]  $\sim^c$  [e, σ]

 $\sim_0$  : ρ0  $\sim^e$  σ0
 $\sim_0$  ()

clo0 : ∀ p → clov p  $\sim^c$  clon p
clo0 p = [p,  $\sim_0$ ]

```

Fig. 7. Agda formalisation of the simulation relation for environment semantics.

```

lemma : cv  $\sim^v$  v → cv  $\sim^c$  cn → v  $\sim$  cn

lemma (var x ρ) [ 'x, ρ $\sim$ σ ]
  with ρ[x]  $\sim^w$   $\sim^v$  σ[x]  $\sim^w$  ← ρ $\sim$ σ x = ρ[x]  $\sim^w$   $\sim^v$  (var σ[x]  $\sim^w$ )

lemma (lam e ρ) [ λ e, ρ $\sim$ σ ] = [ λ e, ρ $\sim$ σ ]  $\sim^v$  (lam e  $\sim$ )

lemma (app [e1, ρ]  $\sim^v$  [λ e, ρ'] [e2, ρ]  $\sim^v$  [e, ρ'▷u]  $\sim^v$ ) [ e1 · e2, ρ $\sim$ σ ]
  with [ λ e, ρ'  $\sim$ σ' ]  $\sim^v$  [e1, σ]  $\sim^v$  [λ e, σ'] ← lemma [e1, ρ]  $\sim^v$  [λ e, ρ'] [ e1, ρ $\sim$ σ ]
  with u  $\sim$  [e2, σ] ← lemma [e2, ρ]  $\sim^v$  [ e2, ρ $\sim$ σ ]
  with v  $\sim^w$   $\sim^v$  [e, σ'▷[e2, σ]]  $\sim^w$  ← lemma [e, ρ'▷u]  $\sim^v$  [ e, ρ'  $\sim$ σ' ▷ u  $\sim$  [e2, σ] ]
  = v  $\sim^w$   $\sim^v$  (app [e1, σ]  $\sim^v$  [λ e, σ'] [e, σ'▷[e2, σ]]  $\sim^w$ )

theorem : clov p  $\sim^v$  v → v  $\sim$  clon p
theorem {p = p} p  $\sim^v$  = lemma p  $\sim^v$  (clo0 p)

```

Fig. 8. The Agda proof of the simulation theorem.

The proof itself, of Lemma 2.1 and equi-termination as its corollary, is presented in Figure 8. By convention, the names of bound variables that represent judgements allude to their types, e.g.,  $[e, \rho' \triangleright u] \sim^v v$  has type  $[e, \rho' \triangleright u] \sim^v v$ . We can observe that it follows the same structure as the paper proof, following by induction on the structure of derivation, making extensive use of ‘with’-notation (McBride & McKinna, 2004) to coordinate pattern matching on recursive calls to `lemma`, corresponding to case analysis on the inductive hypotheses. We believe that the conciseness and clarity of the formalised proof is quite remarkable, and a testament to the simplicity of our proof method, while the ease with which it can be extended to other semantic frameworks and language features speaks to its robustness.

## 5 Conclusion

We have developed and presented a new technique for proving simulation between operational semantics for the same underlying language that differ in eagerness of evaluation. Our approach is elementary, hinging only on definitions of the semantics and (mutually) inductively defined predicates: we believe it is well-suited to early courses of semantics of programming languages, where a simulation theorem like the one we presented can highlight certain properties of semantics, as well as choices and difficulties inherent in proving these theorems. The argument's structure makes it also very natural to mechanise, with the expressive power of Agda paying off particularly handsomely in terms of simplicity in handling of the definitions.

We have also seen that, while the simulation argument is simplest in the big-step, environment-based semantics, the technique scales to other approaches, including both small-step and substitution-based semantics. It is also rather obvious that it scales to calculi with data extending beyond functions (for example, languages that include pairs, numbers or conditional expressions), where the traditional approach would require defining an appropriate reduction theory and proving standardization – a non-trivial task, especially if the evaluation relation of the lazier of the calculi does not naturally coincide with weak-head reduction!

One question that is rather natural to ask is whether our technique is, in some sense, a logical relation (Tait, 1967; Plotkin, 1973; Statman, 1985; Mitchell, 1996, Chapter 8). We believe that it does *not* fit that description: the crucial evidence lies in the treatment of (functional) values. In the logical relations approach, functions or function-like entities are related when they are related *for any related arguments*. In contrast, while the environments saved in a closure do make an appearance in our relation, we still require the bodies of the functions to be *syntactically*, rather than *observationally* equal.

One consequence of this observation is that our technique is unlikely to scale *directly* to a denotational setting (Stoy, 1977; Schmidt, 1986; Winskel, 1993; Reynolds, 1998), where syntax is interpreted away, and the only way to inspect a value's denotation, i.e., a function, is to apply it to an argument. Constructing the simulation in domain theory would inevitably require means markedly more complex than what we advocate in this work, namely recursive (logical) relations over recursive domains (Reynolds, 1974; Pitts, 1993).

If the simplicity of means was not a main concern, one could actually employ syntactic logical relations to build a simulation in the operational setting that would be akin to ours. Such an approach, however, in order to ensure well-formedness of the logical relation, would require the technique of step-indexing (Appel & McAllester, 2001; Ahmed, 2006): since the calculus under consideration is untyped, one has to rely on a different inductive structure than types when defining a logical relation.

A problem pertaining to the one addressed in this note is the relation of call-by-need and call-by-name semantics for lambda-calculus. While proofs of their equivalence have been developed (Ariola & Felleisen, 1997; Maraist *et al.*, 1998; Launchbury, 1993), it would be interesting to see whether our technique scales to this question – and indeed, whether it can serve to simplify or streamline any of the approaches known from the literature. The most promising in this endeavour appears to be Launchbury's call-by-need heap-based natural

semantics that has previously been shown equivalent to a non-deterministic call-by-value semantics, using tools somewhat related to ours (Hackett & Hutton, 2019).

### Data availability statement

The data that support the findings of this study, namely the Agda implementation of the proofs developed in the paper, are openly available at <https://doi.org/10.5281/zenodo.15365178>.

### Acknowledgements

We thank the anonymous reviewers for their helpful comments on the presentation of this work. The first author was supported by the National Science Centre of Poland grant no. 2019/33/B/ST6/00289.

### Supplementary material

The supplementary material for this article can be found at <https://doi.org/10.1017/S0956796825100038>.

### Competing interests

The authors report no conflicts of interest.

### References

- Accattoli, B. & Dal Lago, U. (2012) On the invariance of the unitary cost model for head reduction. In *23rd International Conference on Rewriting Techniques and Applications (RTA '12)*, RTA 2012, May 28–June 2, 2012, Nagoya, Japan, Tiwari, A. (ed), LIPIcs, vol. 15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 22–37. doi: [10.4230/LIPICS.RTA.2012.22](https://doi.org/10.4230/LIPICS.RTA.2012.22).
- Ahmed, A. J. (2006) Step-indexed syntactic logical relations for recursive and quantified types. In *Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2006, Vienna, Austria, March 27–28, 2006, Proceedings*, Sestoft, P. (ed), Lecture Notes in Computer Science, vol. 3924. Springer, pp. 69–83. doi: [10.1007/11693024\\_6](https://doi.org/10.1007/11693024_6).
- Appel, A. W. & McAllester, D. A. (2001) An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.* **23**(5), 657–683. doi: [10.1145/504709.504712](https://doi.org/10.1145/504709.504712).
- Ariola, Z. M. & Felleisen, M. (1997) The call-by-need lambda calculus. *J. Funct. Program.* **7**(3), 265–301.
- Barendregt, H. P. (1985) *The Lambda Calculus – Its Syntax and Semantics*, Studies in Logic and the Foundations of Mathematics, vol. 103. North-Holland.
- Bird, R. S. & Paterson, R. (1999) De Bruijn notation as a nested datatype. *J. Funct. Program.* **9**(1), 77–91.
- Curry, H. B. & Feys, R. (1958) *Combinatory Logic*. Studies in Logic and the Foundations of Mathematics, vol. 1. North-Holland.
- Fiore, M. P., Plotkin, G. D., & Turi, D. (1999) Abstract syntax and variable binding. In *LICS'99*. IEEE Computer Society, pp. 193–202.
- Hackett, J. & Hutton, G. (2019) Call-by-need is clairvoyant call-by-value. *Proc. ACM Program. Lang.* **3**(ICFP), 114:1–114:23. doi: [10.1145/3341718](https://doi.org/10.1145/3341718).
- Harper, R. (2016) *Practical Foundations for Programming Languages*, 2nd ed. Cambridge University Press.

- Kahn, G. (1987) Natural semantics. In STACS'87, Brandenburg, F.-J., Vidal-Naquet, G. & Wirsing, M. (eds). Lecture Notes in Computer Science, vol. 247. Springer, pp. 22–39.
- Landin, P. J. (1964) The mechanical evaluation of expressions. *Comput. J.* **6**(4), 308–320.
- Launchbury, J. (1993) A natural semantics for lazy evaluation. In POPL'93, New York, NY, USA. Association for Computing Machinery, pp. 144–154.
- Maraist, J., Odersky, M. & Wadler, P. (1998) The call-by-need lambda calculus. *J. Funct. Program.* **8**(3), 275–317.
- McBride, C. & McKinna, J. (2004) The view from the left. *J. Funct. Program.* **14**(1), 69–111.
- McKinna, J. & Pollack, R. (1999) Some lambda calculus and type theory formalized. *J. Autom. Reason.* **23**(3–4), 373–409. doi: [10.1023/A:1006294005493](https://doi.org/10.1023/A:1006294005493).
- Mitchell, J. C. (1996) *Foundations for Programming Languages*. Foundation of Computing Series. MIT Press.
- Mitschke, G. (1979) The standardization theorem for  $\lambda$ -calculus. *Math. Log. Q.* **25**(1–2), 29–31. doi: [10.1002/MALQ.19790250104](https://doi.org/10.1002/MALQ.19790250104).
- Norell, U. (2007) *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology.
- Pierce, B. C. (2002) *Types and Programming Languages*. MIT Press.
- Pitts, A. M. (1993) Relational properties of recursively defined domains. In Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS'89), Pacific Grove, California, USA, June 5–8, 1989. IEEE Computer Society, pp. 86–97. doi: [10.1109/LICS.1993.287597](https://doi.org/10.1109/LICS.1993.287597).
- Plotkin, G. D. (1973) *Lambda-Definability and Logical Relations*. Technical Report SAI-RM-4, School of A.I., Univ. of Edinburgh, Edinburgh, Scotland.
- Plotkin, G. D. (1975) Call-by-name, call-by-value and the  $\lambda$ -calculus. *Theor. Comput. Sci.* **1**(2), 125–159.
- Plotkin, G. D. (1981) *A Structural Approach to Operational Semantics*. Technical Report FN-19, DAIMI, Department of Computer Science, Aarhus University, Aarhus, Denmark, September 1981.
- Pollack, R. (1990) Implicit syntax. In Proceedings of the First ESPRIT BRA Workshop on Logical Frameworks, pp. 421–433.
- Reynolds, J. C. (1972) Definitional interpreters for higher-order programming languages. In Proceedings of the ACM Annual Conference, ACM 1972, Volume 2, Donovan, J. J. & Shields, R. (eds). ACM, pp. 717–740.
- Reynolds, J. C. (1974) On the relation between direct and continuation semantics. In Automata, Languages and Programming, 2nd Colloquium, University of Saarbrücken, Germany, July 29–August 2, 1974, Proceedings, Loeckx, J. (ed.). Lecture Notes in Computer Science, vol. 14. Springer, pp. 141–156. doi: [10.1007/3-540-06841-4\\_57](https://doi.org/10.1007/3-540-06841-4_57).
- Reynolds, J. C. (1998) *Theories of Programming Languages*. Cambridge University Press.
- Schmidt, D. A. (1986) *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc.
- Statman, R. (1985) Logical relations and the typed lambda-calculus. *Inf. Control.* **65**(2/3), 85–97.
- Stoy, J. E. (1977) *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press.
- Tait, W. W. (1967) Intensional interpretations of functionals of finite type I. *J. Symb. Log.* **32**(2), 198–212. doi: [10.2307/2271658](https://doi.org/10.2307/2271658).
- Wadler, P., Kokke, W., & Siek, J. G. (2022) *Programming Language Foundations in Agda*. August 2022.
- Wadsworth, C. P. (1971) *Semantics and Pragmatics of the Lambda-Calculus*. PhD thesis, University of Oxford.
- Winskel, G. (1993) *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press.