A Formal Equational Theory for Call-By-Push-Value

Christine Rizkallah[1]*, Dmitri Garbuzov[2], and Steve Zdancewic[2]

[1] University of New South Wales and Data61,CSIRO
[2] University of Pennsylvania **
c.rizkallah@unsw.edu.au, {dmitri, stevez}@cis.upenn.edu

Abstract. Establishing that two programs are contextually equivalent is hard, yet essential for reasoning about semantics preserving program transformations such as compiler optimizations. We adapt Lassen's normal form bisimulations technique to establish the soundness of equational theories for both an untyped call-by-value λ-calculus and a variant of Levy's call-by-push-value language. We demonstrate that our equational theory significantly simplifies the verification of optimizations.

1 Introduction

Establishing program equivalence is a well-known and long-studied problem [15]. Programmers informally think about equivalences when coding: For example, to convince themselves that some refactoring doesn't change the meaning of the program. Tools such as compilers rely on program equivalences when optimizing and transforming code.

Contextual equivalence is the gold standard of what it means for two (potentially open) program terms M_1 and M_2 to be equal. Although the exact technical definition varies from language to language, the intuition is that M_1 is contextually equivalent to M_2 if for every closing context $C[-]$, $C[M_1]$ "behaves the same" as $C[M_2]$. Such contextual equivalences justify program optimizations where we can replace a less-optimal program M_1 by a better M_2 in the program context $C[-]$, without affecting the intended behavior of the program.

While the literature is full of powerful and general techniques for establishing program equivalences using pen-and-paper proofs, not many of these general techniques have been mechanically verified (with some notable exceptions [4]). Moreover, some of these techniques are difficult to apply in practice. For example, *complete* methods, such as applicative bisimulation [1,2], environmental bisimulation [17], and "closed-instances of uses" (CIU) techniques [13], typically require quantification over *all* closed function arguments or closing contexts,

* Work done while at University of Pennsylvania

** This work is supported by NSF grant 1521539. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

which significantly complicates the proofs, especially in the presence of mutually recursive function definitions.

Building on his earlier work [9], Lassen introduced the notion of *eager normal form bisimulations* [10], which is applicable to the call-by-value (CBV) λ-calculus and yet avoids quantification over all arguments or contexts. Lassen's normal form bisimulation $\mathcal{N}$ is *sound* with respect to contextual equivalence but it is not complete—this is the price to be paid for the easier-to-establish equivalences. However, $\mathcal{N}$ is still useful for many equivalence proofs because the relation includes reduction and is a congruence. Lassen defines an equational theory for CBV that is included in $\mathcal{N}$ and hence, it is also sound. This approach is appealing for formal verification because $\mathcal{N}$ is a simple co-inductive relation defined in terms of the operational semantics—mathematical objects that are relatively straightforward to work with in theorem provers.

Here, our aim is to use a similar technique to verify typical compiler optimizations. We introduce a proof structure based on $\mathcal{N}$ and use it to establish the correctness of suitable equational theories. Our development makes a key technical simplification compared to Lassen's work [10]: it does not rely on establishing that $\mathcal{N}$ is itself sound. Instead, we use a variant of $\mathcal{N}$ to prove that two related terms co-terminate. We then directly prove that the equational theory is a congruence, and is therefore sound—we explain this in precise detail below.

We demonstrate our development in two settings. First, we use the untyped call-by-value (CBV) λ-calculus as a familiar vehicle for explaining the ideas in a way that allows for comparison with other approaches. Next, we *scale up* the development to the more complex setting of an untyped variant of Levy's call-by-push-value (CBPV) λ-calculus [12] that includes an explicit letrec construct—this formalization is much more challenging and is our central contribution.

CBPV is a well-known formalism whose metatheory is well-behaved with respect to many extensions. We choose this particular CBPV variant because its features and semantics are closely related to the intermediate languages used by modern compilers [6,7,14]. This makes it attractive for formal verification, since the results can be made applicable to compiler intermediate representations without much extra effort. Our CBPV equational theory is not complete but it includes β-reduction, the operational semantics, and it is a congruence. We show that it nevertheless *trivializes* verifying many typical compiler optimizations.

To summarize, we present sound equational theories for CBV (Section 2) and for a "lower-level" CBPV calculus that includes mutually recursive definitions (Section 3). Our CBPV theory makes it trivial to verify various standard compiler optimizations (Section 4). All of our results are formalized [16] in Coq.

2 The pure untyped call-by-value λ-calculus

In this section, we first demonstrate our proof technique in the case of the untyped call-by-value λ-calculus, which serves as a way to introduce the ideas and as a model of how to proceed in more complex languages.

$$\frac{P\,t}{\mathcal{E}\,P\,t} \quad \frac{\mathcal{E}\,P\,s}{\mathcal{E}\,P\,(s\,t)} \quad \frac{\mathcal{E}\,P\,t}{\mathcal{E}\,P\,(v\,t)} \qquad \frac{}{\mathsf{canon}\ x} \quad \frac{}{\mathsf{canon}\ (\lambda x.\,t)} \quad \frac{\mathcal{E}\,(\lambda s.\,\exists\,x\,v:\ s = x\,v)\,t}{\mathsf{canon}\ t}$$

Fig. 1. Evaluation context closure and canonical forms

The set *Term* of λ-terms are variables (x, y, z), applications, and λ-abstractions:

$$s, t ::= x \quad | \quad s\,t \quad | \quad \lambda x.\,t$$

We identify terms up to α-equivalence. Variables and λ-abstractions are values $\mathcal{V}$ (represented by u and v) and applications are not. If s and t are terms then $s[x := v]$ is defined to be the result of substituting a value v for x in s via capture-avoiding substitution. A term of the form $(\lambda x.s)\,t$ is called a *β-redex*, and has *β-reduct* $s[x := t]$. We say s *β-reduces to* t, written $s \to_\beta t$, if a subterm of s is a β-redex such that t is the result of replacing this subterm by its β-reduct. We define $s \sim_\beta t$ to be the least equivalence relation containing $\to_\beta$. When $s \sim_\beta t$ holds, we say s and t are *β-equivalent*.

Given s and t, these rules define the small step operational semantics $s \longrightarrow t$:

$$\frac{}{(\lambda x.\,t)\,v \longrightarrow t[x := v]} \quad \frac{s_1 \longrightarrow s_2}{s_1\,t \longrightarrow s_2\,t} \quad \frac{t_1 \longrightarrow t_2}{v\,t_1 \longrightarrow v\,t_2}$$

We prove that, as expected, values do not step and that the operational semantics is deterministic. A term t is in *normal form*, written $\mathsf{nf}\ t$, iff $\nexists t' : t \longrightarrow t'$.

2.1 Progress and canonical forms

The bisimulation relation that we will construct relies on relating programs in normal form. We therefore define the predicate canon that holds for terms that are "canonical" in the sense that they are either values or stuck computations. To identify stuck computations, the intuitive idea is to identify terms whose evaluation is blocked because a free variable is in active position.

The set of evaluation contexts for λ-calculus is typically given by the following grammar, where [] is a "hole" indicating where the next evaluation step will occur:

$$E ::= [] \quad | \quad E\,t \quad | \quad v\,E$$

For formalization purposes, we will need to work with terms t of the form $E[s]$, where s (typically a redex) replaces the hole in E. However, rather than reifying evaluation contexts as a datatype and defining corresponding "plugging" and "unique decomposition" operations, we find it more convenient to work with a relational definition of the same concepts.

The higher-order predicate $\mathcal{E}$, whose type is $(\mathit{Term} \to \mathbb{P}) \to \mathit{Term} \to \mathbb{P}$ (where $\mathbb{P}$ is the type of propositions), helps us identify the evaluation context of terms that satisfy a certain property. The $\mathcal{E}$ predicate is defined inductively, as shown in Figure 1. It is parameterized by a proposition P and its structure

$$\frac{R\,s\,t}{\mathcal{P}\,R\,s\,t} \qquad \frac{\mathcal{P}\,R\,s\,t}{\mathcal{P}\,R\,(\lambda x.\,s)\,(\lambda x.\,t)} \qquad \frac{\mathcal{P}\,R\,s\,s'}{\mathcal{P}\,R\,(s\,t)\,(s'\,t)} \qquad \frac{\mathcal{P}\,R\,t\,t'}{\mathcal{P}\,R\,(s\,t)\,(s\,t')}$$

Fig. 2. Linearly compatible closure $\mathcal{P}$ of a (binary) relation R.

mirrors that of the grammar of evaluation contexts. Intuitively, the predicate $\mathcal{E}\,P\,t$ holds if the term t is equal to a term of the form $E[s]$ and $P\,s$ holds. We call $\mathcal{E}\;P$ the *evaluation context closure* of the predicate P.

Using $\mathcal{E}$, it is easy to define stuck computations as the evaluation context closure of an appropriate predicate. The canonical forms predicate canon is defined to hold for values or stuck computations as shown in Figure 1. Note that the expression $\lambda s.\,\exists\,x\,v:\;s = x\,v$ that appears in the third rule is a *meta-level* abstraction: it is a proposition that holds of s when there exists some x and v such that s is of the form $x\,v$ (i.e. s is a variable applied to some value).

We prove that canon is the predicate we want, by showing that it captures (all and only) terms that are in normal form. More formally, we show that for any term t, $\mathsf{nf}\;t \leftrightarrow \mathsf{canon}\;t$. The proof directly follows from the progress theorem which states that for any term t, either $\mathsf{canon}\;t$ holds or $\exists t' : t \longrightarrow t'$ holds.

2.2 Contextual equivalence

Contextual equivalence is the standard way of defining program equality. Two programs s and t are contextually equivalent if $C[s]$ and $C[t]$ either both terminate or both diverge for every closing context $C[-]$. Note that for the untyped λ-calculus in which divergence is the only effect, it is not necessary to explicitly check that they always compute the same result. The intuition is that if there is any situation where they can return different results, one can craft a context in which one terminates and the other diverges. Hence from co-termination in any context one can conclude that they can always be used interchangeably [15].

We define contextual equivalence for CBV. To start, we define a predicate $\mathcal{P}$ that lifts a binary relation R on terms into a relation $\mathcal{P}\;R$ linearly compatible with the λ-calculus syntax, as shown in Figure 2. A relation R that is closed under such a lifting is called *linearly compatible* i.e., if $\forall s\,t : \mathcal{P}\;R\;s\;t \to R\;s\;t$.

We define what it means for terms s and t to appear in the same context $C[-]$ by taking $\mathcal{C}\,s\,t\,s'\,t' \leftrightarrow \mathcal{P}(\lambda\,u\,v.\;u = s \wedge v = t)\;s'\;t'$. We call $(\mathcal{C}\;\;s\;\;t)$ the *contextual closure* of s and t. Intuitively, $\mathcal{C}\;\;s\;\;t\;\;s'\;\;t'$ holds exactly when there exists a context $C[-]$ such that $s' = C[s]$ and $t' = C[t]$. Note that quantifying over all such s' and t' precisely captures the idea of quantifying over all contexts.

A term s *terminates at* t, written $s \Downarrow t$, if $s \longrightarrow^* t \wedge \mathsf{nf}\;t$. Two terms s and t *co-terminate*, written $\mathsf{co\text{-}terminate}\;s\;t$, if $(\exists\,s' : s \Downarrow s') \leftrightarrow (\exists\,t' : t \Downarrow t')$. Two terms s and t are *contextually equivalent* when for all contexts $C[-]$, $C[s]$ halts if and only if $C[t]$ does too. More formally, terms s and t are contextually equivalent, written $s \equiv t$, if $\forall s'\,t' : \mathcal{C}\,s\,t\,s'\,t' \to \mathsf{co\text{-}terminate}\;s'\;t'$. It will be useful to have a separate notion of when a relation R implies co-termination: A relation R is *adequate* if $\forall a\,b :\; R\;a\;b \to \mathsf{co\text{-}terminate}\;a\;b$.

$$\frac{R\,s\,t}{Eq\,R\,s\,t} \qquad \frac{}{Eq\,R\,t\,t} \qquad \frac{Eq\,R\,s\,t}{Eq\,R\,t\,s} \qquad \frac{Eq\,R\,s\,s' \quad Eq\,R\,s'\,t}{Eq\,R\,s\,t}$$

Fig. 3. Equivalence closure of a relation R

$$\frac{}{\dot{\mathcal{P}}\,R\,x\,x} \qquad \frac{R\,s\,s' \;\; \dot{\mathcal{P}}\,R\,s'\,t}{\dot{\mathcal{P}}\,R\,s\,t} \qquad \frac{\dot{\mathcal{P}}\,R\,s\,t}{\dot{\mathcal{P}}\,R\,(\lambda x.\,s)\,(\lambda x.\,t)} \qquad \frac{\dot{\mathcal{P}}\,R\,s\,s' \;\; \dot{\mathcal{P}}\,R\,t\,t'}{\dot{\mathcal{P}}\,R\,(s\,t)\,(s'\,t')}$$

Fig. 4. Alternative formulation of compatible closure $\dot{\mathcal{P}}$.

2.3 Equational theory

With these definitions in mind, we can now define what it means for a program transformation, or more generally for an arbitrary relation, to be sound with respect to contextual equivalence. A term transformation function $f : \mathit{Term} \to \mathit{Term}$ is *sound* if $\forall\, t :\; t \equiv (f\;t)$. A relation R is *sound* if $\forall s\;t :\; R\,s\,t \to s \equiv t$.

To reach our goal of easily verifying various program optimizations, which are often reduction or β-reduction in context, we next develop a sound equational theory. We want an equational theory that is sound (with respect to contextual equivalence) and that includes the operational semantics. Moreover, the equational theory should be a *congruence relation* (i.e. a linearly compatible equivalence relation). Given an equational theory that is congruent and that includes β-reduction, we can conclude that it also includes β-equivalence. Hence, even though it is not complete, it still relates a sufficient number of terms.

Our equational theory simplifies verifying optimizations. We just need to prove that the optimization is included in the equational theory (rather than directly in contextual equivalence). Such a proof can be obtained by verifying and contextually lifting simple transformations—Section 4 gives several examples.

We intuitively want our equational theory to be the congruence closure of the operational step reduction. This way the theory includes the semantics and is a congruence. For terms s and t, a *single-step reduction*, written $s \Rightarrow t$, holds if $\mathcal{P}(\longrightarrow)\;s\;t$. The *equivalence closure* of a relation R, defined using the rules in Figure 3, is the reflexive, symmetric, and transitive closure over R. The equivalence closure of the single-step reduction relation defines the *equational theory for λ-calculus*. Two terms s and t are equal according to the *equational theory for λ-calculus*, written $s \Leftrightarrow t$, if $\mathit{Eq}\,(\Rightarrow)\,s\,t$. It is straightforward to see that our equational theory includes the operational semantics. Proving soundness is significantly more involved and is explained in the next section.

Although the definition of $\mathcal{P}$ given in Figure 2 is a good way to understand the concept of closing a relation under contexts, working with it directly in proofs can be cumbersome. However, because we only care about the equivalence closure of single-step reduction, we can refactor the definitions to build in reflexivity and a bit of transitivity in a way that simplifies the proofs. The resulting variation of $\mathcal{P}$, called $\dot{\mathcal{P}}$, is shown in Figure 4. The equivalence closure of $\dot{\mathcal{P}}$ is the same as that of $\mathcal{P}$ (i.e., $\forall\, R\,s\,t :\; \mathit{Eq}\,(\mathcal{P}\,R)\,s\,t \leftrightarrow \mathit{Eq}\,(\dot{\mathcal{P}}\,R)\,s\,t$), but we no longer need to deal with the nested proof structure needed for the reflexive-transitive closure of

$\mathcal{P}$—instead we can work directly by induction on $\dot{\mathcal{P}}$, which is already reflexive and transitive. We prove that both versions yield the same equational theory.

2.4 Soundness of the equational theory for CBV

Recall that proving that a relation R is *sound* involves proving that terms related by R are contextually equivalent. This is done by proving that R is a linearly compatible relation and by proving that terms related by R co-terminate. Lassen defines a normal form bisimulation relation and uses it to assist in proving soundness of an equational theory for λ-calculus. Similar to Lassen, we also make use of normal form bisimulations in our proof technique. Our proof structure, however, is different than Lassen's. We prove that our equational theory is linearly compatible directly rather than proving that the normal form bisimulation is linearly compatible—we expand on this comparison at the end of this section. This proof follows directly due to the way we define the equational theory, and is, hence, simpler to extend to the more complex call-by-push-value language.

Normal form bisimulation A normal form bisimulation is a bisimulation between executions of terms that either terminate at related normal forms, or diverge. Since program executions can be infinite, normal form bisimulation is defined co-inductively. We first define $\mathcal{N}_s$, which defines one step in the bisimulation, and then use it to define our normal form bisimulation $\mathcal{N}$.

$$\frac{s \longrightarrow^* x \quad t \longrightarrow^* x}{\mathcal{N}_s\, R\, s\, t} \qquad \frac{s \longrightarrow^* \lambda x.\, s' \quad t \longrightarrow^* \lambda x.\, t' \quad R\, s'\, t'}{\mathcal{N}_s\, R\, s\, t}$$

$$\frac{\begin{array}{c} s \longrightarrow^* s' \quad t \longrightarrow^* t' \quad R\, v\, v' \\ \mathcal{E}\,(\lambda t.\, t = x\, v)\, s' \quad \mathcal{E}\,(\lambda t.\, t = x\, v')\, t' \end{array}}{\mathcal{N}_s\, R\, s\, t}\,(*) \qquad \frac{s \longrightarrow^+ s' \quad t \longrightarrow^+ t' \quad R\, s'\, t'}{\mathcal{N}_s\, R\, s\, t}$$

Fig. 5. Normal form bisimulation steps

Figure 5 defines the *normal form bisimulation step* $\mathcal{N}_s$ of a relation R. The normal form bisimulation $\mathcal{N}$ is the greatest relation such that if $\mathcal{N}\, s\, t$ then there exists a relation R such that $\forall s'\, t' :\ R\, s'\, t' \to \mathcal{N}\, s'\, t'$ and $\mathcal{N}_s\, R\, s\, t$.

The *co-induction* lemma for $\mathcal{N}$ intuitively states that to establish $\mathcal{N}$, we need to find a bisimulation relation R that is preserved by $\mathcal{N}_s$. We prove that $\mathcal{N}$ is an equivalence, as we need this in our soundness proof.

Theorem 1. *$\mathcal{N}$ is adequate.* $\forall\, s\, t : \mathcal{N}\, s\, t \to \mathsf{co\text{-}terminate}\ s\ t.$

Congruence of equational theory A relation is a *congruence* if it is a linearly compatible equivalence. By definition, our equational theory is an equivalence.

Theorem 2. *The relation $\Leftrightarrow$ is linearly compatible.*

Proof. By induction on $\mathcal{P}$. The proof heavily relies on the way $\mathcal{P}$ is defined. □

Soundness of the equational theory for CBV

Theorem 3. *$\mathcal{N}$ includes reduction.* $\forall\, s\, t :\ s \Rightarrow t \to \mathcal{N}\, s\, t$.

Theorem 4. *$\mathcal{N}$ includes the equational theory.* $\forall\, s\, t :\ s \Leftrightarrow t \to \mathcal{N}\, s\, t$.

Theorem 5. *The equational theory is sound.* $\forall\, s\, t :\ s \Leftrightarrow t \to s \equiv t$.

Proof. Given $s \Leftrightarrow t$, we want to show that $\forall s'\, t' : \mathcal{C}\, s\, t\, s'\, t' \to \mathsf{co\text{-}terminate}\ s'\ t'$. For any terms s' and t' such that $\mathcal{C}\, s\, t\, s'\, t'$, we know $\mathcal{P}(\lambda\, u\, v.\ u = s \wedge v = t)\ s'\ t'$ by definition of $\mathcal{C}$. From $s \Leftrightarrow t$ and $\mathcal{P}(\lambda\, u\, v.\ u = s \wedge v = t)\ s'\ t'$ we can infer $\mathcal{P}(\Leftrightarrow)\ s'\ t'$. Since $\Leftrightarrow$ is linearly compatible (Theorem 2) we know $s' \Leftrightarrow t'$. Since $\mathcal{N}$ includes $\Leftrightarrow$ (Theorem 4) it follows that $\mathcal{N}\, s'\, t'$, and by adequacy (Theorem 1), we can conclude $\mathsf{co\text{-}terminate}\ s'\ t'$. □

Comparison Similar to Lassen we make use of $\mathcal{N}$ but the structure of our soundness proof differs. Lassen defines $\mathcal{N}$ and proves that it is sound. He proves that his equational theory is included in $\mathcal{N}$ and is, therefore, also sound.

We show that our equational theory is sound by directly showing it is linearly compatible and only using $\mathcal{N}$ to assist in proving that the equational theory is adequate. In fact, our CBV version of $\mathcal{N}$ is adequate yet unsound for contextual equivalence. It relates $(\lambda y.y)(x\lambda z.z)$ and $(\lambda y.\Omega)(x\lambda z.z)$ which are both stuck on x (as $\mathcal{N}_s$'s $(*)$ rule allows the evaluation contexts on each side to be unrelated), and the context $(\lambda x.[-])\lambda u.u$ distinguishes them. Our proof method does not rely on $\mathcal{N}$ being sound and this simpler definition suffices for proving adequacy.

We decided to take this approach because the way we define our equational theory makes proving that it is a linearly compatible relation entirely straightforward. The definition of $\mathcal{N}$ is designed to ease the adequacy proof.

3 A call-by-push-value language

We show how the same essential proof structure can be applied to a more complex language to establish the soundness of its equational theory. We choose a variant of Levy's call-by-push-value (CBPV) language as our target because it is a "low-level" language with a rich equational theory. We redefine and reuse some of the notation that was introduced for CBV in the context of CBPV.

3.1 Syntax

The top of Figure 6 shows the syntax for our variant of Levy's call-by-push-value calculus (CBPV) [12], which serves as the basis for our definitions. CBPV is a somewhat lower level and more structured functional language than the ordinary λ-calculus. The key feature is that it distinguishes *values* from *computations*. Values include variables x, natural numbers n, and suspended computations $\mathsf{thunk}\, M$, whereas computations include: $\mathsf{force}\, V$, which runs a suspended thunk;

$$
\begin{array}{rrcl|l|l}
\text{Values} & \ni V & ::= & x & n & \mathsf{thunk}\, M \\
\text{Terms} & \ni M, N & ::= & \mathsf{force}\, V & & \mathsf{letrec}\, x_1 = M_1, \ldots, x_n = M_n\, \mathsf{in}\, N \\
& & | & \mathsf{prd}\, V & & M\, \mathsf{to}\, x\, \mathsf{in}\, N \\
& & | & V \cdot M & & \lambda x.M \\
& & | & V_1 \oplus V_2 & & \mathsf{if0}\ V\ M_1\ M_2 \\
\text{Sorts} & \ni S & ::= & \mathsf{V} & \mathsf{C} &
\end{array}
$$

$$
\frac{}{M \rightsquigarrow M} \qquad \frac{\{\, \overline{\mathsf{thunk}\,(\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, M_i)/x_i}^{\,i} \,\}\, N \rightsquigarrow N'}{\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, N \rightsquigarrow N'}
$$

$$
\frac{}{\mathsf{force}\,(\mathsf{thunk}\, M) \to M} \qquad \frac{}{\mathsf{if0}\ 0\ M_1\ M_2 \to M_1} \qquad \frac{}{\mathsf{if0}\ n\ M_1\ M_2 \to M_2}\ (n \neq 0)
$$

$$
\frac{M \rightsquigarrow \mathsf{prd}\, V}{M\, \mathsf{to}\, x\, \mathsf{in}\, N \to \{V/x\}\, N} \qquad \frac{M \rightsquigarrow \lambda x.N}{V \cdot M \to \{V/x\}\, N} \qquad \frac{M \to M'}{V \cdot M \to V \cdot M'}
$$

$$
\frac{M \to M'}{M\, \mathsf{to}\, x\, \mathsf{in}\, N \to M'\, \mathsf{to}\, x\, \mathsf{in}\, N} \qquad \frac{M \rightsquigarrow (n_1 \oplus n_2)}{M\, \mathsf{to}\, x\, \mathsf{in}\, N \to \{n_1 [\![\oplus]\!] n_2/x\}\, N}
$$

$$
\frac{N \rightsquigarrow N' \quad N' \to M}{N \to M} \qquad \frac{\{\, \overline{\mathsf{thunk}\,(\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, M_i)/x_i}^{\,i} \,\}\, N \longrightarrow N'}{\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, N \longrightarrow N'}
$$

$$
\frac{}{\mathsf{wf_V}\ x} \qquad \frac{}{\mathsf{wf_V}\ n} \qquad \frac{\mathsf{wf_C}\ M}{\mathsf{wf_V}\ (\mathsf{thunk}\, M)} \qquad \frac{\mathsf{wf_V}\ V}{\mathsf{wf_C}\ (\mathsf{prd}\ V)} \qquad \frac{\mathsf{wf_C}\ M \quad \mathsf{wf_C}\ N}{\mathsf{wf_C}\ (M\, \mathsf{to}\, x\, \mathsf{in}\, N)}
$$

$$
\frac{\overline{\mathsf{wf_C}\ M}^{\,i} \qquad \mathsf{wf_C}\ N}{\mathsf{wf_C}\ (\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, N)} \qquad \frac{\mathsf{wf_C}\ M}{\mathsf{wf_C}\ (\lambda x.M)} \qquad \frac{\mathsf{wf_V}\ V}{\mathsf{wf_C}\ (\mathsf{force}\ V)}
$$

$$
\frac{\mathsf{wf_V}\ V \quad \mathsf{wf_C}\ M}{\mathsf{wf_C}\ (V \cdot M)} \qquad \frac{\mathsf{wf_V}\ V_1 \quad \mathsf{wf_V}\ V_2}{\mathsf{wf_C}\ (V_1 \oplus V_2)} \qquad \frac{\mathsf{wf_V}\ V \quad \mathsf{wf_C}\ M_1 \quad \mathsf{wf_C}\ M_2}{\mathsf{wf_C}\ (\mathsf{if0}\ V\ M_1\ M_2)}
$$

Fig. 6. Syntax, operational semantics, and wellformedness for the CBPV language.

mutually recursive definitions, $\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, M$; monadically-structured sequences of computations, written $M\, \mathsf{to}\, x\, \mathsf{in}\, N$, which runs M to produce a computation of the form $\mathsf{prd}\ V$ and then binds V as x in N; λ-abstraction $\lambda x.M$ and application $V \cdot M$ (V is the *argument* and M is the *function*); binary arithmetic operations $V_1 \oplus V_2$, where $\oplus$ is addition, subtraction, or less-than; and finally, conditional statements $\mathsf{if0}\ V\ M_1\ M_2$ that run M_1 if V is 0, otherwise, run M_2.

The most difficult aspect of formalizing this language is dealing with the $\mathsf{letrec}\, \overline{x_i = M_i}^{\,i}\, \mathsf{in}\, M$ form. The intended semantics of this term is that each of the x_i's is bound in all of the M_i's and in M. As a consequence, we have to define a multiway substitution operation, which we denote $\{\, \overline{V_i/x_i}^{\,i} \,\}\, M$. It means the simultaneous substitution of each V_i for x_i in M. Our Coq code uses de Bruijn indices, but we present the language with named variables for better readability.

3.2 Wellformedness

In Levy's presentation, the value and computation terms are separated *syntactically*, with distinct grammars for each. In our Coq formulation, we have found it simpler to combine both syntaxes into one recursive definition, which avoids combining nested recursion (for the lists of bindings found in letrec) with mutual recursion. As a consequence, we separately define (mutually recursive) wellformedness predicates that distinguish values from computations; we say values have sort V and computations have sort C.

A term M is wellformed when there exists a sort S such that $\mathsf{wf}_S\ M$ according to the rules in Figure 6. Thanks to this separation, many of our definitions later on are adapted to account for an extra condition parameter in order to only account for wellformed terms. As a matter of notation, we use the metavariable V to mean values and M, M', N, etc. to mean computations.

In another departure from Levy's original presentation of CBPV, our version is *untyped*. Because most of our results pertain to the dynamic semantics of the language, we have eschewed types here; however, incorporating a type system should be a fairly straightforward adaptation of our formalism. In particular, the parameters necessary to account for wellformedness can simply be instantiated with a typing predicate instead.

3.3 Structural operational semantics

The value-computation distinction is a key feature of the CBPV design: its evaluation order is completely determined thanks to restrictions that ensure there is never a choice between a substitution step and a congruence rule. The middle portion of Figure 6 gives the details of the operational semantics, whose small-step evaluation relation is denoted by $M \longrightarrow M'$.

Once again, the real challenge for our formalization is how to deal with letrec. The usual approach is to allow the structural operational semantics to unroll a letrec as a step of computation, which, in our setting would amount to using the following rule: $\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ M \longrightarrow \{\ \overline{\mathsf{thunk}\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ M_i)/x_i}^{\,i}\ \}\ M$. Note that because the variables x_i range over values but the right-hand sides of the letrec bindings are computations M_i, we have to wrap each computation in a thunk during the unrolling. However, rather than using this rule directly, we have opted to construct the operational semantics in such a way that letrec unrolling doesn't "count" as an operational step. Therefore the operational semantics rules in Figure 6 rely on an auxiliary relation $\rightsquigarrow$ that unrolls a letrec to expose either a prd or a λ term. The operational semantics rules are otherwise straightforward and consist of three "real" steps of computation: forcing a thunk, sequencing, and β-substitution, and three congruence rules that search for the next redex. Our choice to handle letrec in this way is not strictly necessary (our techniques would apply with the "standard" interpretation above); however, we prefer this formulation, despite its slight cost in complexity, because we anticipate that treating letrec as having no runtime cost is more consistent with the semantics of low-level compiler intermediate representations [7].

Given the definitions of the wellformedness relation and operational semantics, it is easy to establish some basic facts. A term M is in *normal form*, written $\mathsf{nf}\ M$, if $\nexists M' : M \longrightarrow M'$. We prove that the step relation is deterministic and hence each term has at most one normal form:

- For terms M, N, and N', if $M \longrightarrow N$ and $M \longrightarrow N'$, then $N = N'$.
- For terms M, N, and N', if $M \longrightarrow^* N$, $M \longrightarrow^* N'$, $\mathsf{nf}\ N$, and $\mathsf{nf}\ N'$, then $N = N'$.

The step relation also preserves wellformedness (note that values do not step):

- For terms M and M', if $\mathsf{wf_C}\ M$ and $M \longrightarrow M'$, then $\mathsf{wf_C}\ M'$.
- For terms M and M', if $\mathsf{wf_C}\ M$ and $M \longrightarrow^* M'$, then $\mathsf{wf_C}\ M'$.

3.4 Progress, and canonical and error forms

Similar to CBV, the bisimulation relation for CBPV also relies on relating terms in normal form However, for CBPV there are also certain *erroneous terms*, that are not considered to be "good" CBPV programs and that cannot step. In this pure setting, such programs could be ruled out by using a type system, but similar issues arise if we extended the language with nontrivial constants and partial operations (such as division or array lookup). We therefore define two predicates error and canon, which partition normal terms into two sets: those that we consider "erroneous" and those that are "canonical" in the sense that they are good CBPV terms that are nevertheless stuck. Canonical terms arise because we want to be able to relate *open* terms. For example, $(\mathsf{force}\, x)\ \mathsf{to}\, x'\ \mathsf{in}\, M$ cannot step because it is blocked on trying to evaluate the term $\mathsf{force}\, x$.

To define these predicates, the intuitive idea would be to identify terms whose evaluation is blocked because the next step of computation must force a variable (or, perform an ill-typed action). Setting aside letrec for the moment, the evaluation contexts for our CBPV language are given by the following grammar:

$$E ::= [\,] \quad | \quad E\ \mathsf{to}\, x\ \mathsf{in}\, N \quad | \quad V \cdot E$$

We want terms of the form $E[\mathsf{force}\, x]$ to be canonical, and, similarly, terms like $E[(\lambda x.M)\ \mathsf{to}\, x'\ \mathsf{in}\, N]$ to be erroneous. However, as our operational semantics treats letrec as transparent, we must account for unrolling of recursive definitions.

We introduce a higher-order predicate $\mathcal{E}$ whose type is $(\mathit{Term} \to \mathbb{P}) \to \mathit{Term} \to \mathbb{P}$ (similar to the one we defined for the CBV λ-calculus). $\mathcal{E}$ is defined inductively, as shown in Figure 7. It is parameterized by a proposition P and its structure mirrors that of the grammar of evaluation contexts, except that it also builds in a case for unrolling letrec. Intuitively, the predicate $\mathcal{E}\ P\ M$ holds if the term M unrolls to a term of the form $E[M']$ and $P(M')$ holds. We call $\mathcal{E}\ P$ the *evaluation context closure* of the predicate P.

Using $\mathcal{E}$, it is straightforward to define $\mathsf{error}\ M$ as the evaluation context closure of a predicate errorP that picks out the ill-formed terms. Similarly, $\mathsf{canon}\ M$ instantiates $\mathcal{E}$ with the predicate $\mathsf{forcevar}$, which holds of a term M exactly when M is $\mathsf{force}\, x$ for some variable x. We prove that these predicates form a partition of wellformed computations (note that values do not step):

$$\frac{P\,(M)}{\mathcal{E}\;P\;M} \qquad \frac{\mathcal{E}\;P\;M}{\mathcal{E}\;P\;(M\;\mathsf{to}\;x\;\mathsf{in}\;N)} \qquad \frac{\mathcal{E}\;P\;M}{\mathcal{E}\;P\;(V\cdot M)}$$

$$\frac{\mathcal{E}\;P\;(\{\;\overline{\mathsf{thunk}\;(\mathsf{letrec}\;\overline{x_i = M_i}^{\;i}\;\mathsf{in}\;M_i)/x_i}^{\;i}\;\}\;N)}{\mathcal{E}\;P\;(\mathsf{letrec}\;\overline{x_i = M_i}^{\;i}\;\mathsf{in}\;N)}$$

$$\frac{M \rightsquigarrow \lambda x.M'}{\mathsf{errorP}\;(M\;\mathsf{to}\;x\;\mathsf{in}\;N)} \qquad \frac{M \rightsquigarrow \mathsf{prd}\;V'}{\mathsf{errorP}\;(V\cdot M)} \qquad \mathsf{error}M \Leftrightarrow \mathcal{E}\;\mathsf{errorP}\;M$$

$$\frac{M \rightsquigarrow \mathsf{prd}\;V}{\mathsf{canon}\;M} \qquad \frac{M \rightsquigarrow \lambda x.M'}{\mathsf{canon}\;M} \qquad \frac{\mathcal{E}\;\mathsf{forcevar}\;M}{\mathsf{canon}\;M} \qquad \mathsf{forcevar}M \leftrightarrow (\exists x.\; M = \mathsf{force}\;x)$$

Fig. 7. Evaluation context closure, and canonical and error forms

- For a term M, if $\mathsf{wf_C}\;M$ and $\mathsf{canon}\;M$, then $\mathsf{nf}\;M \wedge \neg\mathsf{error}\;M$.
- For a term M, if $\mathsf{wf_C}\;M$ and $\mathsf{error}\;M$, then $\mathsf{nf}\;M \wedge \neg\mathsf{canon}\;M$.
- For a term M, if $\mathsf{wf_C}\;M$, then $\mathsf{canon}\;M$ or $\mathsf{error}\;M$ or $\exists\,N : M \longrightarrow N$.

3.5 Contextual equivalence

Recall that contextual equivalence equates two terms if their behavior is the same in all program contexts. In the context of CBPV, we are only interested in reasoning about wellformed terms. We therefore define a *conditional linearly compatible closure* that restricts the context to that of terms which respect a condition. The condition is later used to restrict the context to wellformed terms.

We define *conditional linearly compatible closure*, $\mathcal{P}$, similar to that of CBV but with an additional condition P. It lifts a relation R into a relation $\mathcal{P}\;R\;P$ linearly compatible with the CBPV syntax and that relates terms for which P holds (defined in Appendix A). A relation that is closed under such lifting for wellformed terms is called *wellformed linearly compatible*. Formally, a relation R on terms is *wellformed linearly compatible* if $\forall M\;N : \mathcal{P}\;R\;(\lambda x.\;\exists\;S : \mathsf{wf}_S\;x)\;M\;N \rightarrow R\;M\;N$. The conditional contextual closure $\mathcal{C}$ is defined as follows:

$$\mathcal{C}\;P\;M\;N\;M'\;N' \;\leftrightarrow\; \mathcal{P}(\lambda\,u\,v.\;u = M \wedge v = N)\;P\;M'\;N'$$

Intuitively, $\mathcal{C}\;P\;M\;N\;M'\;N'$ holds exactly when there exists a context $C[-]$ such that $M' = C[M]$, $N' = C[N]$, $P\;(C[M])$, and $P\;(C[N])$. Instantiating P with $(\lambda x.\;\exists\;S : \mathsf{wf}_S\;x)$ and quantifying over all M' and N' captures the idea of quantifying over all contexts for wellformed terms.

A term M *terminates to* N, written $M \Downarrow N$, if $M \longrightarrow^* N \wedge \mathsf{nf}\;N$. Two terms M and N *co-terminate*, written $\mathsf{co\text{-}terminate}\;M\;N$, if $(\exists\,M' : M \Downarrow M') \leftrightarrow (\exists\,N' : N \Downarrow N')$. Two terms M and N are *contextually equivalent*, written $M \equiv N$, if $\forall M'\;N' : \mathcal{C}\;(\lambda x.\;\exists\;S : \mathsf{wf}_S\;x)\;M\;N\;M'\;N' \rightarrow \mathsf{co\text{-}terminate}\;M'\;N'$. The definitions of soundness and adequacy are identical to the ones in Section 2 except that they operate on CBPV terms.

$$\frac{P\ M}{Eq\ R\ P\ M\ M} \qquad \frac{P\ M \quad (R\ M\ M' \vee R\ M'\ M) \quad Eq\ R\ P\ M'\ N}{Eq\ R\ P\ M\ N}$$

Fig. 8. Conditional equivalence closure of a relation R over a predicate P

3.6 Equational theory

The equational theory for CBPV is defined in a similar fashion to that for CBV. Once again, we are interested in a theory that is sound and that includes the operational semantics. As mentioned earlier, also ensuring that the equational theory is a congruence relation results in a theory that includes β-equivalence. Once again all our definitions in the CBPV context are parameterized with a condition, which is in turn used to limit our scope to wellformed terms.

A relation R is *conditionally reflexive* on a predicate P if $\forall x : P\,x \to R\,x\,x$. A relation R is *conditionally symmetric* on a predicate P if $\forall x\,y : P\,x \to P\,y \to R\,x\,y \to R\,y\,x$. A relation R is *conditionally transitive* on a predicate P if $\forall\, x\,y\,z : P\,x \to P\,y \to P\,z \to R\,x\,y \to R\,y\,z \to R\,x\,z$. A relation R is a *conditional equivalence* on a predicate P if it is conditionally reflexive, conditionally symmetric, and conditionally transitive on P.

Our CBPV operational semantics does not explicitly unroll letrec as a step of computation. Nevertheless, we would like our equational theory to equate letrec terms to their unrolled version. Therefore, we first define a reduction relation.

Definition 1. *Given two terms M and N, the reduction relation $M \dashrightarrow N$ is defined using the following two rules:*

$$\frac{M \longrightarrow N}{M \dashrightarrow N} \qquad \frac{}{\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ M \dashrightarrow \{\ \overline{\mathsf{thunk}\ (\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ M_i)/x_i}^{\,i}\ \}\ M}$$

Definition 2. *For wellformed terms M and N, the* parallel reduction *relation, written $M \Rightarrow N$, holds if $\mathcal{P}(\dashrightarrow)\ M\ N$.*

The conditional equivalence closure of a relation R over a predicate P is the reflexive, symmetric, and transitive closure of R where P holds. It is defined using the rules in Figure 8 where the second rule combines symmetry and transitivity.

Lemma 1. *For any two elements x and y related by the conditional equivalence closure of a relation R on P, $P\,x$ holds and $P\,y$ holds.*

Lemma 2. *Given two relations R and R', and given a predicate P, if $\forall x'\,y' : P\,x' \to P\,y' \to R\,x'\,y' \to R'\,x'\,y'$ and R' is a conditional equivalence on P, then for any two elements x and y related using the conditional equivalence closure of R on P, $R'\,x\,y$ holds.*

Two terms M and N are equal according to the *equational theory for CBPV*, written $M \Leftrightarrow_S N$, if $Eq\ (\Rightarrow)\ \mathsf{wf}_S\ M\ N$. Once again, it is straightforward to see that our equational theory includes the operational semantics. We present the soundness proof of the equational theory for CBPV in the next section.

$$\frac{}{\mathcal{N}_s\, R\, \mathsf{V}\, x\, x} \qquad \frac{}{\mathcal{N}_s\, R\, \mathsf{V}\, n\, n} \qquad \frac{R\, \mathsf{C}\, M\, N}{\mathcal{N}_s\, R\, \mathsf{V}\, (\mathsf{thunk}\, M)\, (\mathsf{thunk}\, N)} \qquad \frac{M \longrightarrow^* M' \;\; N \longrightarrow^* N' \quad \mathsf{error}\; M' \;\; \mathsf{error}\; N'}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

$$\frac{M \longrightarrow^* M' \;\; N \longrightarrow^* N' \;\; M' \rightsquigarrow \mathsf{prd}\; V \quad N' \rightsquigarrow \mathsf{prd}\; V' \;\; R\, \mathsf{V}\; V\; V'}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N} \qquad \frac{M \longrightarrow^* M' \;\; N \longrightarrow^* N' \;\; M' \rightsquigarrow \lambda x.M'' \quad N' \rightsquigarrow \lambda x.N'' \;\; R\, \mathsf{C}\, M''\, N''}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

$$\frac{M \longrightarrow^* M' \;\; N \longrightarrow^* N' \quad \mathcal{E}_{\mathcal{C}}\, R\, (\mathsf{force}\, x)\, (\mathsf{force}\, x)\, M'\, N'}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N} \qquad \frac{\begin{array}{c} M \longrightarrow^* M' \;\; N \longrightarrow^* N' \\ \mathcal{E}_{\mathcal{C}}\, R\, (\mathsf{if0}\; x\; M_1\; M_2)\, (\mathsf{if0}\; x\; M_1'\; M_2')\, M'\, N' \\ R\, \mathsf{C}\, M_1\, M_1' \quad R\, \mathsf{C}\, M_2\, M_2' \end{array}}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

$$\frac{\begin{array}{c} M \longrightarrow^* M' \;\; N \longrightarrow^* N' \\ M' \rightsquigarrow x \oplus V \;\; N' \rightsquigarrow x \oplus V' \\ R\, \mathsf{V}\; V\; V' \quad (\nexists\, M'' : V = \mathsf{thunk}\, M'') \end{array}}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N} \qquad \frac{\begin{array}{c} M \longrightarrow^* M' \;\; N \longrightarrow^* N' \\ M' \rightsquigarrow n \oplus V \;\; N' \rightsquigarrow n \oplus V' \\ R\, \mathsf{V}\; V\; V' \quad (\nexists\, M'' : V = \mathsf{thunk}\, M'') \end{array}}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

$$\frac{\begin{array}{c} M \longrightarrow^* M' \quad N \longrightarrow^* N' \quad \mathcal{E}_{\mathcal{C}}\, R\, (M_1\, \mathsf{to}\, x\, \mathsf{in}\, M_2)\, (N_1\, \mathsf{to}\, x\, \mathsf{in}\, N_2)\, M'\, N' \\ M_1 \rightsquigarrow V_1 \oplus V_2 \quad N_1 \rightsquigarrow V_1' \oplus V_2' \quad R\, \mathsf{V}\; V_1\; V_1' \quad R\, \mathsf{V}\; V_2\; V_2' \quad R\, \mathsf{C}\, M_2\, N_2 \\ ((V_1 = x \wedge (\nexists M : V_2 = \mathsf{thunk}\, M)) \vee (V_2 = x \wedge (\nexists M : V_1 = \mathsf{thunk}\, M))) \end{array}}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

$$\frac{M \longrightarrow^+ M' \;\; N \longrightarrow^+ N' \;\; R\, \mathsf{C}\, M'\, N'}{\mathcal{N}_s\, R\, \mathsf{C}\, M\, N}$$

Fig. 9. Normal form bisimulation steps

3.7 Soundness of the equational theory for CBPV

The soundness proof of the equational theory for CBPV follows the same structure as that for CBV. The proof development is restricted to wellformed terms.

Normal form bisimulation Similar to CBV, $\mathcal{N}_s$ defines one step in the bisimulation for CBPV and is used to co-inductively define our bisimulation $\mathcal{N}$ for CBPV. One difference to the CBV definition is that $\mathcal{N}$ also takes a sort as input and relates two terms at that sort. Moreover, the definition relies on $\mathcal{E}_{\mathcal{C}}$, which lifts a binary relation R across two terms and ensures that they share an evaluation context up to R. Namely, $\mathcal{E}_{\mathcal{C}}\, R\, M\, N\, M'\, N'$ holds if $R\, M'\, N'$ and if $M'\; N'$ are evaluation contexts over M and N, respectively.

Figure 9 defines the *normal form bisimulation step* $\mathcal{N}_s$ of a relation R (that given a sort relates terms). The normal form bisimulation $\mathcal{N}$ is the greatest relation such that if $\mathcal{N}\, S\, M\, N$ then there exists a relation R such that $\forall S'\, M'\, N' : R\, S'\, M'\, N' \rightarrow \mathcal{N}\, S'\, M'\, N'$ and $\mathcal{N}_s\, R\, S\, M\, N$.

Theorem 6. *$\mathcal{N}$ is a conditional equivalence on wellformed terms.*

Congruence of equational theory A relation is called a *congruence* if it is both an equivalence over wellformed terms and wellformed linearly compatible. By definition, our equational theory is an equivalence over wellformed terms.

Theorem 7. *The relation* $(\lambda M\ N : \exists\, S : M \Leftrightarrow_S N)$ *is wellformed linearly compatible.*

Soundness of equational theory for CBPV

Theorem 8. $\mathcal{N}$ *includes parallel reduction*

$$\forall\, S\ M\ N : \mathsf{wf}_S\ M \to \mathsf{wf}_S\ N \to M \Rightarrow N \to \mathcal{N}\ S\ M\ N$$

Theorem 9. $\mathcal{N}$ *includes the equational theory*

$$\forall\, S\ M\ N :\ M \Leftrightarrow_S N \to \mathcal{N}\ S\ M\ N$$

Proof. Follows directly from Lemma 2 along with Theorems 6 and 8. □

Theorem 10. $\mathcal{N}$ *is adequate*

$$\forall\, S\ M\ N :\ \mathsf{wf}_S\ M \to \mathsf{wf}_S\ N \to \mathcal{N}\ S\ M\ N \to \mathsf{co\text{-}terminate}\ M\ N$$

We only prove and rely on the adequacy and equivalence of $\mathcal{N}$. Although unnecessary, we suspect our CBPV $\mathcal{N}$ is a congruence, thus, is sound. Unlike CBV, $\mathcal{N}$ here uses $\mathcal{E}_{\mathcal{C}}$ which, unlike $\mathcal{E}$, only relates terms that share an evaluation context.

Theorem 11. *The equational theory is sound.* $\forall\, S\ M\ N :\ M \Leftrightarrow_S N \to M \equiv N$.

Proof. Given $M \Leftrightarrow_S N$, we want to show

$$\forall M'\ N' : \mathcal{C}\ (\lambda x.\ \exists\ S : \mathsf{wf}_S\ x)\ M\ N\ M'\ N' \to \mathsf{co\text{-}terminate}\ M'\ N'.$$

For any terms M' and N' such that $\mathcal{C}\ (\lambda x.\ \exists\ S : \mathsf{wf}_S\ x)\ M\ N\ M'\ N'$, we know $\mathcal{P}(\lambda\, u\, v.\ u = M \wedge v = N)\ (\lambda x.\ \exists\ S : \mathsf{wf}_S\ x)\ M'\ N'$ by definition of $\mathcal{C}$. From $M \Leftrightarrow_S N$ and $\mathcal{P}(\lambda\, u\, v.\ u = M \wedge v = N)\ (\lambda x.\ \exists\ S : \mathsf{wf}_S\ x)\ M'\ N'$ we can infer $\mathcal{P}\ (\lambda M\ N : \exists S : M \Leftrightarrow_S N)\ (\lambda x.\ \exists\ S : \mathsf{wf}_S\ x)\ M'\ N'$. Since $(\lambda M\ N : \exists S : M \Leftrightarrow_S N)$ is linearly compatible (Theorem 7) we know $M' \Leftrightarrow_{S'} N'$ for some sort S'. From Lemma 1 we know $\mathsf{wf}_{S'}\ M'$ and $\mathsf{wf}_{S'}\ N'$. Since $\mathcal{N}$ includes the equational theory (Theorem 9) it follows that $\mathcal{N}\ S'\ M'\ N'$. Since $\mathcal{N}$ is adequate (Theorem 10) we conclude that $\mathsf{co\text{-}terminate}\ M'\ N'$. □

4 Verifying optimizations using our equational theory

Compilers rely on program equivalences when optimizing and transforming programs. In the context of verified compilation, as found in the CompCert [11] and CakeML [8] projects, formal verification of particular program equivalences is crucial: the correctness of classic optimizations like constant folding, code inlining, loop unrolling, etc., hinges on such proofs. Therefore, techniques that facilitate such machine-checked proofs have the potential for a broad impact.

We demonstrate how our CBPV equational theory makes it trivial to verify many typical compiler optimizations and indicate which imperative optimizations correspond to our results.

Connection to compilers for imperative languages In separate work [7], we prove an equivalence between our functional CBPV language and *control flow graphs* (CFG), which many compilers for imperative languages use to represent low-level programs. Control flow graphs are suited to program analysis and optimizations. However, formalizing the behavior and metatheory of CFG programs is non-trivial: CFG programs don't compose well, their semantics depends on auxiliary state, and, as a consequence, they do not enjoy a simple equational theory that can be used for reasoning about the correctness of program transformations. The equational theory developed in this paper can also be used to reason about CFG optimizations.

Optimizations Figure 10 summarizes various desirable CBPV compiler optimizations that are easily proven sound using our equational theory by term rewriting. We give a short C description of the optimizations before explaining how we verified them in Coq.

CBPV equation	optimization
$\mathsf{force}\,(\mathsf{thunk}\,M) \equiv M$	block merging "direct jump case"
$V \cdot \lambda x.M \equiv \{V/x\}\,M$	block merging "phi case"
$\mathsf{prd}\ V\ \mathsf{to}\ x\ \mathsf{in}\ M \equiv \{V/x\}\,M$	move elimination
$(n_1 \oplus n_2)\ \mathsf{to}\ x\ \mathsf{in}\ M \equiv \{n_1 [\![\oplus]\!] n_2/x\}\,M$	constant folding
$\mathsf{thunk}\ (\lambda y.M) \cdot \lambda x.N \equiv \{\mathsf{thunk}\ (\lambda y.M)/x\}\,N$	function inlining
$\mathsf{if0}\ 0\ M_1\ M_2 \equiv M_1$	dead branch elimination "true branch"
$\mathsf{if0}\ n\ M_1\ M_2 \equiv M_2$ where $(n \neq 0)$	dead branch elimination "false branch"
$\mathsf{if0}\ V\ M\ M \equiv M$	branch elimination

Fig. 10. CBPV equations and corresponding imperative optimizations

Block merging "direct jump case": merges two blocks of the control-flow-graph if the first is the unique predecessor of the second and jumps directly.

Block merging "phi case": replaces functions applied to arguments with their result (does one step of β-reduction). At the CFG level, this optimization corresponds to eliminating a jump to a block containing a "phi" node.

Move elimination: eliminates a move by substituting the target register's value.

Function inlining: replaces function calls in the program with the bodies of the called functions. Note that the function inlining CBPV equation is just an instance of the block merging "phi case" equation.

Dead branch elimination: removes dead branches in conditional statements.

Branch elimination: replaces conditional statements that have identical branches with the code of one of the branches. Unlike the three previous optimizations, this one does not follow directly from one step in the operational semantics.

However, it still falls in our equational theory and it was easy to verify it using an additional case analysis on V in the proof.

Proof technique We define a function mk-cmp that takes an optimization function $f : \mathit{Term} \to \mathit{Term}$ and a program M and returns an optimized program M' where the optimization function f has been applied recursively throughout the program (first on all subprograms then again recursively on the result). The equations in Figure 10 are examples of such functions f. Given the term on the left-hand side of the equation as input, f returns the right-hand side; given any other term, f acts as the identity. Applying mk-cmp to these functions applies them throughout the program repeatedly and results in the actual optimizations that we verify.

We first prove two general theorems about mk-cmp, one about preserving wellformedness and the other about preserving soundness.

Theorem 12. mk-cmp *preserves wellformedness*

$$\begin{aligned}\forall f\, S\, M : (\forall\, S'\, N : \mathsf{wf}_{S'}\ N \to \mathsf{wf}_{S'}\,(f\ N)) \to \\ \mathsf{wf}_S\ M \to \mathsf{wf}_S\,(\mathsf{mk\text{-}cmp}\, f\ M)\end{aligned}$$

This theorem intuitively states that if f preserves wellformedness then so does mk-cmp f.

Theorem 13. mk-cmp *preserves soundness*

$$\begin{aligned}\forall f\, S\, M : (\forall\, S'\, N : \mathsf{wf}_{S'}\ N \to \mathsf{wf}_{S'}\,(f\ N)) \to \\ (\forall\, S'\, N : \mathsf{wf}_{S'}\ N \to N \equiv (f\ N)) \to \\ \mathsf{wf}_S\ M\ \to\ M \equiv (\mathsf{mk\text{-}cmp}\, f\ M)\end{aligned}$$

This theorem states that if f preserves wellformedness and is sound then mk-cmp f is also sound.

Corollary 1. mk-cmp *preserves soundness using the CBPV equational theory*

$$\begin{aligned}\forall f\, S\, M : (\forall\, S'\, N : \mathsf{wf}_{S'}\ N \to \mathsf{wf}_{S'}\,(f\ N)) \to \\ (\forall\, S'\, N : \mathsf{wf}_{S'}\ N \to N \Leftrightarrow_{S'} (f\ N)) \to \\ \mathsf{wf}_S\ M\ \to\ M \equiv (\mathsf{mk\text{-}cmp}\, f\ M)\end{aligned}$$

This corollary states that if f preserves wellformedness and is included in the CBPV equational theory, then mk-cmp f is sound. This corollary follows directly from Theorem 13 along with the fact that the equational theory is sound (Theorem 11).

The proof structure for the optimizations mk-cmp f, where f is any of the eight optimizations described in Figure 10, proceeds as follows:
First, we prove that f preserves wellformedness:

$$\forall\, S\, M : \mathsf{wf}_S\ M \to \mathsf{wf}_S\,(f\, M).$$

Then, we prove that the equational theory includes f:

$$\forall\, S\, M : \mathsf{wf}_S\ M \to M \Leftrightarrow_S (f\ M).$$

Finally, using the mk-cmp corollary (Corollary 1) along with these results we can conclude that the optimization is correct:

$$\forall\, S\, M : \mathsf{wf}_S\ M \to M \equiv (\mathsf{mk\text{-}cmp}\ f\ M).$$

Our CBPV equational theory made it simple to prove all these optimizations correct in Coq with a minimal amount of effort. This demonstrates the power of our approach in reducing the cost of verifying compiler optimizations. The same approach will similarly facilitate reasoning about more complex optimizations.

5 Conclusion and future work

We developed a sound equational theory for a variant of Levy's low-level CBPV language and showed how it makes verifying several typical optimizations trivial. For the sake of explaining and comparing our proof method, we also applied it to the pure untyped CBV λ-calculus. Similar to prior work on proving equivalences using $\mathcal{N}$ we did not target completeness (w.r.t contextual equivalence). We rather focused on sound reasoning techniques that are ideal for verified compilers.

The adequacy of reduction, that we rely on in our soundness proof, can alternatively be derived from the Standardization Theorem for λ-calculus [18,5], which states that there is a "standard" reduction sequence for any multi-step reduction [3]. Takahashi gave a simple proof of the Standardization Theorem for call-by-name [18] and Crary adapted and formalized her proof for call-by-value [5]. We plan to investigate whether adapting and scaling this proof method to our CBPV language, which allows mutual recursion, provides a simpler proof than using $\mathcal{N}$.

References

1. Abramsky, S.: The lazy λ- calculus. In: Research topics in functional programming, pp. 65–116. Addison Wesley (1990)
2. Abramsky, S., Ong, C.H.L.: Full abstraction in the lazy lambda calculus. Information and Computation **105**(2), 159–267 (1993)
3. Barendregt, H.: The Lambda Calculus: Its Syntax and Semantics. Studies in Logic and the Foundations of Mathematics, Elsevier Science (2013)
4. Biernacki, D., Lenglet, S., Polesiuk, P.: Proving soundness of extensional normal-form bisimilarities. In: Mathematical Foundations of Program Semantics (MFPS). Electronic Notes in Theoretical Computer Science (2017)
5. Crary, K.: A simple proof of call-by-value standardization. Technical Report CMU-CS-09-137, Carnegie Mellon University (2009), `https://www.cs.cmu.edu/~crary/papers/2009/standard.pdf`
6. Downen, P., Maurer, L., Ariola, Z.M., Jones, S.P.: Sequent calculus as a compiler intermediate language. In: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP). pp. 74–88 (2016)

7. Garbuzov, D., Mansky, W., Rizkallah, C., Zdancewic, S.: Structural operational semantics for control flow graph machines. CoRR **abs/1805.05400** (2018), http://arxiv.org/abs/1805.05400
8. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: Cakeml: A verified implementation of ml. In: Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 179–191. POPL '14, ACM, New York, NY, USA (2014). https://doi.org/10.1145/2535838.2535841, http://doi.acm.org/10.1145/2535838.2535841
9. Lassen, S.B.: Bisimulation in untyped lambda calculus: Böhm trees and bisimulation up to context. Electronic Notes in Theoretical Computer Science **20**, 346–374 (1999)
10. Lassen, S.B.: Eager Normal Form Bisimulation. In: 20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05). pp. 345–354 (2005)
11. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning **43**(4), 363–446 (2009). https://doi.org/10.1007/s10817-009-9155-4
12. Levy, P.B.: Call-by-push-value: A subsuming paradigm. In: Girard, J. (ed.) Typed Lambda Calculi and Applications, 4th International Conference, TLCA'99, L'Aquila, Italy, April 7-9, 1999, Proceedings. Lecture Notes in Computer Science, vol. 1581, pp. 228–242. Springer (1999)
13. Mason, I., Talcott, C.: Equivalence in functional languages with effects. Journal of functional programming **1**(3), 287–327 (1991)
14. Maurer, L., Downen, P., Ariola, Z.M., Peyton Jones, S.: Compiling without continuations. In: Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 482–494. ACM (2017)
15. Morris, J.H.: Lambda Calculus Models of Programming Languages. Ph.D. thesis, Massachusets Institute of Technology (1968)
16. Rizkallah, C., Garbuzov, D., Zdancewic, S.: (2018), http://www.cse.unsw.edu.au/~crizkallah/publications/equational_theory_cbpv.tar.gz, accompanying Coq formalization.
17. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on. pp. 293–302. IEEE (2007)
18. Takahashi, M.: Parallel reductions in lambda-calculus. Inf. Comput. **118**(1), 120–127 (1995). https://doi.org/10.1006/inco.1995.1057, https://doi.org/10.1006/inco.1995.1057

A Appendix: Definition of $\mathcal{P}$ for CBPV

$$\frac{P\ M \quad P\ N \quad R\ M\ N}{\mathcal{P}\ R\ P\ M\ N} \qquad \frac{P\,(\mathsf{prd}\ V) \quad P\,(\mathsf{prd}\ V) \quad \mathcal{P}\ R\ P\ V\ V'}{\mathcal{P}\ R\ P\,(\mathsf{prd}\ V)\,(\mathsf{prd}\ V')}$$

$$\frac{P\,(\mathsf{force}\ V) \quad P\,(\mathsf{force}\ V') \quad \mathcal{P}\ R\ P\ V\ V'}{\mathcal{P}\ R\ P\,(\mathsf{force}\ V)\,(\mathsf{force}\ V')} \qquad \frac{P\,(\lambda x.M) \quad P\,(\lambda x.N) \quad \mathcal{P}\ R\ P\ M\ N}{\mathcal{P}\ R\ P\,(\lambda x.M)\,(\lambda x.N)}$$

$$\frac{P\,(\mathsf{thunk}\ M) \quad P\,(\mathsf{thunk}\ N) \quad \mathcal{P}\ R\ P\ M\ N}{\mathcal{P}\ R\ P\,(\mathsf{thunk}\ M)\,(\mathsf{thunk}\ N)}$$

$$\frac{P\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N) \quad P\,(M'\ \mathsf{to}\ x\ \mathsf{in}\ N) \quad \mathcal{P}\ R\ P\ M\ M'}{\mathcal{P}\ R\ P\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N)\,(M'\ \mathsf{to}\ x\ \mathsf{in}\ N)}$$

$$\frac{P\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N) \quad P\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N') \quad \mathcal{P}\ R\ P\ N\ N'}{\mathcal{P}\ R\ P\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N)\,(M\ \mathsf{to}\ x\ \mathsf{in}\ N')}$$

$$\frac{P\,(V \cdot M) \quad P\,(V' \cdot M) \quad \mathcal{P}\ R\ P\ V\ V'}{\mathcal{P}\ R\ P\,(V \cdot M)\,(V' \cdot M)} \qquad \frac{P\,(V \cdot M) \quad P\,(V \cdot N) \quad \mathcal{P}\ R\ P\ M\ N}{\mathcal{P}\ R\ P\,(V \cdot M)\,(V \cdot N)}$$

$$\frac{P\,(V_1 \oplus V_2) \quad P\,(V_1' \oplus V_2) \quad \mathcal{P}\ R\ P\ V_1\ V_1'}{\mathcal{P}\ R\ P\,(V_1 \oplus V_2)\,(V_1' \oplus V_2)} \qquad \frac{P\,(V_1 \oplus V_2) \quad P\,(V_1 \oplus V_2') \quad \mathcal{P}\ R\ P\ V_2\ V_2'}{\mathcal{P}\ R\ P\,(V_1 \oplus V_2)\,(V_1 \oplus V_2')}$$

$$\frac{P\,(\mathsf{if0}\ V\ M_1\ M_2) \quad P\,(\mathsf{if0}\ V'\ M_1\ M_2) \quad \mathcal{P}\ R\ P\ V\ V'}{\mathcal{P}\ R\ P\,(\mathsf{if0}\ V\ M_1\ M_2)\,(\mathsf{if0}\ V'\ M_1\ M_2)}$$

$$\frac{P\,(\mathsf{if0}\ V\ M_1\ M_2) \quad P\,(\mathsf{if0}\ V\ M_1'\ M_2) \quad \mathcal{P}\ R\ P\ M_1\ M_1'}{\mathcal{P}\ R\ P\,(\mathsf{if0}\ V\ M_1\ M_2)\,(\mathsf{if0}\ V\ M_1'\ M_2)}$$

$$\frac{P\,(\mathsf{if0}\ V\ M_1\ M_2) \quad P\,(\mathsf{if0}\ V\ M_1\ M_2') \quad \mathcal{P}\ R\ P\ M_2\ M_2'}{\mathcal{P}\ R\ P\,(\mathsf{if0}\ V\ M_1\ M_2)\,(\mathsf{if0}\ V\ M_1\ M_2')}$$

$$\frac{P\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N) \quad P\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N') \quad \mathcal{P}\ R\ P\ N\ N'}{\mathcal{P}\ R\ P\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N)\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N')}$$

$$\frac{\begin{array}{c}P\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N) \quad P\,(\mathsf{letrec}\ \overline{x_i = M_i'}^{\,i}\ \mathsf{in}\ N) \\ \mathcal{P}\ R\ P\ M_j\ M_j' \text{ where } j \text{ is one of the } i\text{'s}\end{array}}{\mathcal{P}\ R\ P\,(\mathsf{letrec}\ \overline{x_i = M_i}^{\,i}\ \mathsf{in}\ N)\,(\mathsf{letrec}\ \overline{x_i = M_i'}^{\,i}\ \mathsf{in}\ N)}$$

Fig. 11. Rules defining conditional compatible closure $\mathcal{P}$ of a (binary) relation R and a (unary) predicate P.