On the Operational Theory of the CPS-Calculus: Towards a Theoretical Foundation for IRs

PAULO TORRENS*, University of Kent, United Kingdom
DOMINIC ORCHARD, University of Kent, United Kingdom and University of Cambridge, United Kingdom
CRISTIANO VASCONCELLOS, Santa Catarina State University, Brazil

The continuation-passing style translation often employed by compilers gives rise to a class of intermediate representation languages where functions are not allowed to return anymore. Though the primary use of these intermediate representation languages is to expose details about a program's control flow, they may be equipped with an equational theory in order to be seen as specialized calculi, which in turn may be related to the original languages by means of a factorization theorem. In this paper, we explore Thielecke's CPS-calculus, a small theory of continuations inspired by compiler implementations, and study its metatheory. We extend it with a sound reduction semantics that faithfully represents optimization rules used in actual compilers, and prove that it acts as a suitable theoretical foundation for the intermediate representation of Appel's and Kennedy's compilers by following the guidelines set out by Plotkin. Finally, we prove that the CPS-calculus is strongly normalizing in the simply typed setting by using a novel proof method for reasoning about reducibility at a distance, from which logical consistency follows. Taken together, these results close a gap in the existing literature, providing a formal theory for reasoning about intermediate representations.

CCS Concepts: • **Software and its engineering** → **General programming languages**; **Compilers**; • **Theory of computation** → *Type theory*.

Additional Key Words and Phrases: Continuations, intermediate representations, strong normalization.

ACM Reference Format:
Paulo Torrens, Dominic Orchard, and Cristiano Vasconcellos. 2024. On the Operational Theory of the CPS-Calculus: Towards a Theoretical Foundation for IRs. *Proc. ACM Program. Lang.* 8, ICFP, Article 241 (August 2024), 30 pages. https://doi.org/10.1145/3674630

1 Introduction

The concept of a *continuation* is anything but new to computer scientists. Ever since their many discoveries [Reynolds 1993], continuations have been widely studied and have proven themselves useful for many different purposes, theoretical and practical alike. In the realm of compilers, the *continuation-passing style* (CPS) translation bears great importance as a program transformation technique and has found its way into several production compilers, especially when trying to extract the imperative program lying underneath a functional one.

Source code is most commonly written in the so-called *direct style* (DS): there is a notion of a call stack and, upon termination, a function returns to the point where it was called, resuming any remaining work. The CPS translation aims to turn a program in DS into an equivalent program in

*The first half of this work has been carried during the first author's time working at Santa Catarina State University.

Authors' Contact Information: Paulo Torrens, University of Kent, Canterbury, United Kingdom, paulotorrens@gnu.org; Dominic Orchard, University of Kent, Canterbury, United Kingdom and University of Cambridge, Cambridge, United Kingdom, d.a.orchard@kent.ac.uk; Cristiano Vasconcellos, Santa Catarina State University, Joinville, Brazil, cristiano.vasconcellos@udesc.br.

ACM 2475-1421/2024/8-ART241
https://doi.org/10.1145/3674630

CPS by internalizing the context in which every function call appears: it gives the context explicitly as an extra parameter, thus removing the need for a call stack. In this process, every call becomes a tail call, evaluation contexts themselves become trivial, and functions need not return anymore. By doing so, many details of the source program are exposed, including the evaluation order which is made explicit and intermediate results which become named.

Such details are of remarkably great importance during compilation. So, in order to explore these details in the process of *lowering* a program written in some high-level programming language into a low-level target language, compilers employ *intermediate representation* (IR) languages, which are designed specifically for this task. There are, of course, different styles of IRs, but, as pointed out in the pioneering work of Steele [1978], a program in CPS is not only well-suited for optimization but it also allows for a very natural imperative interpretation by seeing calls to continuations as *jumping*, indeed making the transformed program closer to machine code. This makes a *CPS language*, in which all programs are written in CPS, a strong candidate for a good IR.

Reasoning about programs in CPS. It is well-known that the λ-calculus is both the prototypical functional programming language and a suitable theoretical foundation for studying functional languages. While it is common to use the λ-calculus to reason about CPS terms if they are to be closed under β-reduction [Flanagan et al. 1993], this is not always the case. For example, the IRs used by Appel [1991] and Kennedy [2007] require that every value is named and referenced through a variable. While the image of some common CPS translations have this property, β-reduction breaks it. Even though these kind of languages have been used in production compilers for a long time now (e.g., in SML/NJ and SML.net), they still lack a common theoretical foundation, relying on either repeatedly reinventing the wheel or using ad hoc methods. Ideally, there should be some theoretical foundation for reasoning about such IRs that would relate to them in the same sense that the λ-calculus relates to a functional language. We are left to ask how to properly demonstrate that they are related. Fortunately, there is already a well-trodden path for doing so set by Plotkin [1975], who demonstrated the relationship between the λ_v-calculus and ISWIM. In this paper, we follow Plotkin's program as a guideline to establish a foundation for name-passing IRs.

As a motivating example which has inspired this paper, even though we do not explore this subject further in it, consider the study of *type-preserving compilation* [Morrisett et al. 1999]. It has been noted that preserving the types from the source program into the IR is desirable as it allows for more optimization opportunities and may be used to achieve link-time safety [Bowman et al. 2017]; however, compilation typically targets either the λ-calculus or some new, custom IR designed for the purpose. Trying to take a similar approach for an IR arising from existing practical implementations, such as Appel's, may not be so simple, though. For example, it is not clear how to prove properties such as subject reduction or normalization for these IRs as there is no established notion of reduction semantics for them. Our thesis is that such investigations are better served by having a well-understood foundation with well-established results such as one finds, e.g., with the λ-calculus and the π-calculus, but which are not yet established for common IRs.

However, if we are to explore this idea, then we must first consider: what is a prototypical IR? We recall that CPS translations are generally a technique for embedding one language into another. For example, in yet another one of its discoveries, a CPS translation was used to encode the λ-calculus into the π-calculus [Milner 1992]. Given that translations may be given to several different targets, one may wonder: what exactly is the structure necessary for interpreting CPS terms? This question led to categorical interpretations of continuation-passing and CPS translations [Führmann and Thielecke 2004] and to Thielecke's *CPS-calculus* [Thielecke 1997b], which we study here. If we can relate the CPS-calculus to an IR in the same way as Plotkin did for the λ-calculus and programming languages, then this demonstrates that it can be used as a theoretical foundation for such IRs.

Contributions. We summarize our contributions as follows:

- We adapt the evaluator given by Kennedy [2007] as the operational semantics for his IR, and demonstrate that computation in it matches the one derived by essential steps in the CPS-calculus. As a consequence, behavioral equivalences in both coincide.
- We give a new reduction semantics to the CPS-calculus based on rewriting rules used in compilers, accommodating both computational steps and optimizations through a notion of shrinking reduction steps, and prove that it is sound.
- We demonstrate that the CPS-calculus is a computationally adequate compilation target by showing that it can simulate evaluation for the λ-calculus by using Plotkin's CPS translations, and explore how this may be used for proving optimizations correct.
- We investigate our new reduction semantics, showing it to be confluent by introducing a theory of residuals, and showing that it allows for the factorization of essential steps, from which follows that the CPS-calculus is related to Appel's IR as expected.
- As a last step, we study the simply typed CPS-calculus, for which we prove type soundness and type preservation. We also show that reduction is strongly normalizing for well-typed terms, which requires adapting the well-known reducibility proof method by showing that semantic types are preserved under the usual structural typing rules.

Additionally, most results presented in this paper have been at least partially mechanically verified in the Coq proof assistant, having nearly 22,000 lines of code as of writing. This is still a work in progress, with the remaining proofs having been written on pen and paper and currently in the process of being formalized (notably Theorems 2.1 and 4.5 and Lemmas 7.5 to 7.7). At the moment, some results still rely on admitted properties (most of which are technical in nature), but they work in increasing our confidence in the proofs presented in this paper. Therefore, we delegate most of the technical details to the source code, which is available on GitHub[1] under a permissive free software license, and to its accompanying documentation. Though this is left as future work, we aim that this may be used as the basis for the implementation of a verified optimizing compiler.

Proceeding, we will start in Section 1.1 by briefly expanding on the required background. Section 2 reviews the CPS-calculus and existing results, serving to fix its syntax and key components used in the rest of the paper. Our novel contributions begin with Section 3, where we adapt the machine semantics and propose the novel reduction semantics. In Sections 4 to 7 we study the metatheory of the CPS-calculus in presence of such semantics, and in Section 8 we conclude.

Notation. We first set some conventions used throughout this paper. For clarity, we write terms in the CPS-calculus in blue and sans-serif, terms in the λ-calculi in **red and bold**, and terms in the π-calculus in *pink and italic*, following the suggestions by Patrignani [2020]. We also take the set of variables in each calculus to be distinct but isomorphic, so, for example, for any variable x in the CPS-calculus, there is a corresponding variable **x** in the λ-calculi. We represent a possibly empty, comma separated list of some syntactic object by writing a vector arrow above them, such that $\vec{x}$ represents a list $x_1, ..., x_n$, and freely treat such lists as a set and distribute operations pointwise over them. Any kind of association list, such as a typing environment, is treated as a function from variables to its mapped set. For any relation $\mathcal{R}$, we write $\mathcal{R}^?$, $\mathcal{R}^+$ and $\mathcal{R}^*$ for its reflexive, transitive, and reflexive and transitive closures respectively, and we freely denote relation composition by juxtaposition. For any given syntactic class, we reserve the same symbol as a metavariable ranging over such class, as well as the immediately following symbols in alphabetical order as required.

[1]The formal development is available at https://github.com/takanuva/cps. A copy is also archived on Zenodo, available at https://zenodo.org/doi/10.5281/zenodo.11498450 [Torrens 2024].

1.1 Background

CPS languages and the λ-calculi. What exactly qualifies a CPS language? It is not enough that we consider only that a language may be used as the target for a CPS translation, as that would be too broad. For example, several CPS translations from the λ-calculus onto itself have been proposed in the literature. Instead, we focus on those languages that are only large enough to accommodate a CPS translation such that they do not allow functions in direct style (DS) anymore, instead providing only a way to encode them. In order to illustrate this idea, let us recall the definitions of the call-by-name (CBN) and call-by-value (CBV) λ-calculi, as given by Plotkin [1975]:

$$\begin{array}{lll} \textit{Expressions} & e & ::= \; v \mid e\ e \\ \textit{Values} & v & ::= \; x \mid \lambda x.e \\ \textit{Contexts} & C & ::= \; [-] \mid C\ e \mid e\ C \mid \lambda x.C \end{array} \qquad \begin{array}{lll} \textit{Continuations} & r & ::= \; k \mid \lambda x.s \\ \textit{Trivial exprs.} & t & ::= \; x \mid \lambda x.\lambda k.s \\ \textit{Serious exprs.} & s & ::= \; r\ t \mid t\ t\ r \mid (\lambda k.s)\ r \end{array}$$

$\boxed{e \to_\beta e}$ *Reduction for the λ-calculus*

$$(\beta) \quad (\lambda x.e)\ f \;\to_\beta\; e[f/x]$$

$\boxed{e \to_{\beta_v} e}$ *Reduction for the λ_v-calculus*

$$(\beta_v) \quad (\lambda x.e)\ v \;\to_{\beta_v}\; e[v/x]$$

$\boxed{e \to_n e}$ *Call-by-name reduction*

$$\frac{}{(\lambda x.e)\ f \;\to_n\; e[f/x]} \qquad \frac{e_1 \to_n e_2}{e_1\ f \to_n e_2\ f} \qquad \frac{e_1 \to_n e_2}{x\ e_1 \to_n x\ e_2}$$

$\boxed{e \to_v e}$ *Call-by-value reduction*

$$\frac{}{(\lambda x.e)\ v \;\to_v\; e[v/x]} \qquad \frac{e_1 \to_v e_2}{e_1\ f \to_v e_2\ f} \qquad \frac{e_1 \to_v e_2}{v\ e_1 \to_v v\ e_2}$$

We consider Church's λ-calculus as a programming language, which we call λ_n (cf. Abramsky [1990]), by taking $\to_n$ above as its semantics.[2] Similarly, we consider Plotkin's λ_v-calculus as a programming language, which we call λ_v, by taking $\to_v$ as its semantics, while noticing that both relations are deterministic (i.e., they are partial functions). As several CPS translations exist, consider for now the CBV translation given by Plotkin [1975], which will be formally defined in Section 4. As it has been noted, a simple way to define a CPS language is to take the image of the CPS translation under consideration and close it under some desired notion of reduction [Sabry and Wadler 1997]. For Plotkin's CBV translation, a DS term e is turned into some CPS term s [Sangiorgi and Walker 2001], which belongs to the strict subset of the λ-calculus terms able to encode the CBV evaluation order in CPS, as characterized above. We could then define a CPS language for it by taking either the CBN or the CBV reduction as its semantics given that, by the indifference lemma [Plotkin 1975], they coincide for terms strictly in CPS.

Plotkin's program. One of our main goals is to demonstrate that the CPS-calculus works as a foundation for Appel's and Kennedy's IRs, but how exactly can this be stated? In his seminal work, Plotkin [1975] investigates a similar question and shows how to relate a calculus and a programming language:[3] namely, a language is considered correct with regard to a calculus if its evaluator can be properly mimicked by the calculus' theory, and a calculus is considered correct with regard to a language if it can only equate terms that cannot be distinguished within the language.

[2] Although the third CBN rule in $\to_n$ is often omitted, we include it as it was originally defined, where it was necessary for the indifference result.

[3] Informally, we consider a calculus as an equational theory and a programming language as an operational semantics given through an abstract machine or an interpreter for the same syntax (cf. Ariola et al. [1995]).

Plotkin's paper sets out a *program* of work for establishing such a relationship: he defines the theory of the λ_v-calculus by taking the (β_v) rule and treating it symmetrically as the only axiom of its equational theory, and shows that $\to_{\mathrm{v}}$ emulates the execution of the SECD machine used to define the semantics of Landin's ISWIM, resulting in a program reducing to a value only when execution under the machine terminates with the same value. Plotkin gives a reduction semantics for the calculus using $\to_{\beta_{\mathrm{v}}}$, which is the contextual closure of the (β_v) rule, and proves that it is confluent and that reduction may always be reordered to start with $\to_{\mathrm{v}}$, from which the correspondence follows. He does the same thing for the standard λ-calculus and λ_n, using, respectively, the (β) rule as an axiom, $\to_\beta$ (also contextual) as the reduction relation, and $\to_{\mathrm{n}}$ as the evaluator. This same route has also been used to study the call-by-need λ-calculus by Ariola et al. [1995].

Digression: on the choice for an IR. While considering the concept of a prototypical IR, we must point out the different successful choices that have been used in production compilers, such as the static single assignment (SSA) form [Cytron et al. 1991], which is the go-to choice for compiling imperative languages, and A-normal form (ANF) [Flanagan et al. 1993], both of which resemble code in DS. In fact, there has been a long debate on what exactly constitutes a good IR and whether a CPS language is the best choice. Although we do not wish to contribute to this discussion, we refer the reader to the works of Kennedy [2007], Maurer et al. [2017], and Cong et al. [2019].

While Appel's compiler allows the use of control effects, such as Lisp's `call/cc` or Felleisen's $\mathcal{C}$ operator, it has been argued by Kennedy [2007] that a CPS language might still be useful in the absence of these. His IR, based on Appel's, presents a syntactic distinction between source variables and continuations, which is enough to prevent control effects and to make such a language equivalent to SSA [Appel 1998; Kelsey 1995] and also to ANF and Moggi's λ_c-calculus [Sabry and Felleisen 1992], the latter of which has also been used as an IR [Benton et al. 1998; Kennedy 2007]. In a way, we could argue that the equivalence between those calculi suggests that an important aspect of a good IR is its monadic structure: evaluation must be sound in the presence of any possible side effects, something that doesn't happen in the λ-calculus, and which rules it out as a proper foundation for IRs in its usual presentation.

We thus justify our choice of the CPS-calculus, which allows control effects just as Appel's IR, as being the more general setting, where a simple syntactic restriction would be enough to keep the language equivalent to SSA or ANF if desired. While the explicit use of continuations may be seen as a downside, we note that, as argued by Cong et al. [2019], most stock hardware actually works in CPS by storing the return address as if it were a continuation, meaning that it's not a question of *whether* to add explicit continuations in a compiler pipeline or not, but rather *when* to do so. While we remark this as an important and relevant issue, we leave the formal investigation about the relationship between the CPS-calculus and idealized versions of SSA and ANF as future work.

2 The CPS-calculus

The CPS-calculus is a simple name-passing calculus, inspired by and based upon the IR of Appel's compiler [Appel 1991]. It was originally introduced by Thielecke [1997b], and further explored in his PhD thesis [Thielecke 1997a], where it was used to aid in the study of $\otimes\neg$-categories.[4] It may also be seen as "a common idiom for the λ-calculus and the π-calculus" [Thielecke 1997a], as it is "the distilled target of CPS translations" [Merro 2010] presented as a calculus in its own right. Compared to Appel's IR, it drops non-essential features such as integers and recursion (which can be derived in the untyped setting), in a similar fashion to how λ_v captures the core components of ISWIM. In this section, we briefly review existing results on its theory.

[4]Pronounced as "tensor-not categories", as confirmed by Paul Blain Levy: https://mathoverflow.net/q/363297/142280.

2.1 Syntax

Though some variants of the CPS-calculus were introduced by Thielecke [1997a], we will focus on the polyadic, non-linear and non-recursive variant, which has just enough structure to be used as a compilation target (cf. Kennedy [2007]). Its syntax is defined as follows:

$$\textit{Commands}\quad \mathsf{b} \;::=\; \underbrace{\mathsf{x}\langle\vec{\mathsf{x}}\rangle}_{\textit{jump}} \;\mid\; \underbrace{\mathsf{b}\ \{\ \mathsf{x}\langle\vec{\mathsf{x}}\rangle = \mathsf{b}\ \}}_{\textit{bind}}$$

A CPS term, also called a *command*, is defined as either a *jump* or as a *bind*. In the former, $\mathsf{k}\langle\vec{\mathsf{x}}\rangle$ represents a call to k with $\vec{\mathsf{x}}$ as arguments, and in the latter, $\mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}$ composes two terms by defining k in b as a continuation c, which will take $\vec{\mathsf{y}}$ as parameters. In the syntax of Appel and Jim [1997], these terms would be written respectively as $\mathtt{k}(\vec{\mathtt{x}})$ and $\mathtt{let\ k}(\vec{\mathtt{y}})\ \mathtt{=\ c\ in\ b}$, which might be more familiar to some readers. However, due to the more theoretical nature of this work, closer in spirit to Thielecke's, we rather stick with his syntax, reserving Appel's for an actual in-practice IR.

Terms will always be identified up to renaming of bound variables[5] and as such any relation on terms said to be reflexive will contain α-convertibility. We write $\mathsf{c}[\vec{\mathsf{x}}/\vec{\mathsf{y}}]$ for the capture-avoiding parallel substitution of variables $\vec{\mathsf{y}}$ free in c by variables $\vec{\mathsf{x}}$, defined in a standard way, noting that we always make the implicit assumption that $\vec{\mathsf{x}}$ and $\vec{\mathsf{y}}$ are lists of variables with the same size (otherwise the substitution is undefined). We also write $\mathsf{FV}(\mathsf{c})$ for the set of free variables in c and, as usual, say a term is *closed* if it has no free variables.

Before proceeding to formally define the semantics of the calculus, and in order to first illustrate the intended meaning of terms, consider the following two translations, given by Thielecke [1997b], respectively into the λ-calculus and the π-calculus:

$\boxed{\mathsf{b}^{\circ} = \mathsf{e}}$ *Translation into the λ-calculus*

$$\begin{aligned} \mathsf{k}\langle\vec{\mathsf{x}}\rangle^{\circ} &= \mathsf{k}\ \vec{\mathsf{x}} \\ \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{x}}\rangle = \mathsf{c}\ \}^{\circ} &= (\lambda \mathsf{k}.\mathsf{b}^{\circ})\ (\lambda\vec{\mathsf{x}}.\mathsf{c}^{\circ}) \end{aligned}$$

$\boxed{\mathsf{b}^{\bullet} = p}$ *Translation into the π-calculus*

$$\begin{aligned} \mathsf{k}\langle\vec{\mathsf{x}}\rangle^{\bullet} &= \overline{k}\langle\vec{x}\rangle \\ \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{x}}\rangle = \mathsf{c}\ \}^{\bullet} &= (\nu k)(\mathsf{b}^{\bullet} \mid !k(\vec{x}).\mathsf{c}^{\bullet}) \\ & \qquad (\text{w.l.o.g., } \mathsf{k} \notin \mathsf{FV}(\mathsf{c})) \end{aligned}$$

Notice that the latter maps terms into the localized, asynchronous variant of the π-calculus: only asynchronous outputs are used in the translation, and, as input always happens inside of a local environment [Pierce and Sangiorgi 1993], the input capability cannot be transmitted.

2.2 Equational Theory

Just as in the λ-calculi and the π-calculus, the CPS-calculus had its semantics originally given through an equational theory, also called the axiomatic semantics by Merro [2010]. The equational theory is defined through a set of axioms that form a contextual equivalence relation, allowing one to derive equality between terms that "behave the same". In order to define a contextual relation, we must first define what constitutes a *context*, which will play an important role in defining the formal meaning of terms.

The set of contexts is generated by the following grammar:

$$\textit{Contexts}\quad \mathsf{C} \;::=\; \underbrace{[-] \;\mid\; \mathsf{C}\ \{\ \mathsf{x}\langle\vec{\mathsf{x}}\rangle = \mathsf{b}\ \}}_{\textit{static}} \;\mid\; \mathsf{b}\ \{\ \mathsf{x}\langle\vec{\mathsf{x}}\rangle = \mathsf{C}\ \}$$

[5] Formally, we use the convention of Barendregt [1981], i.e., that in every mathematical context, free and bound variables are always assumed to be distinct. Note this also implies that every bound variable has to be different from each other.

The special symbol $[-]$, called a *hole*, represents a single missing subterm in a context. We write $C[c]$ to denote the action of *filling* a context, which replaces the hole in a context C with the term c (allowing for name capture). A context which is generated by the first two context productions alone is called a *static context* [Merro 2010], where the hole appears only in the leftmost position, which we also call the *head position*. A relation on terms is said to be *compatible* if it is preserved by arbitrary contexts, and a *congruence* if it is both compatible and an equivalence. We also write $\mathsf{dom}(C)$ for the set of variables that the context binds in its hole, where:

$$\mathsf{dom}([-]) = \emptyset \qquad \mathsf{dom}(C\ \{\ k\langle\vec{y}\rangle = c\ \}) = \{k\} \cup \mathsf{dom}(C) \qquad \mathsf{dom}(b\ \{\ k\langle\vec{y}\rangle = C\ \}) = \{\vec{y}\} \cup \mathsf{dom}(C)$$

The equational theory for the CPS calculus may be properly defined now as the least congruence closed under the following four axioms (from Thielecke [1997b]):

$\boxed{b = b}$ *Equational Theory*

$$\begin{array}{lrcll}
\text{(JMP)} & k\langle\vec{x}\rangle\ \{\ k\langle\vec{y}\rangle = c\ \} & = & c[\vec{x}/\vec{y}] & (\text{given } k \notin \vec{x}) \\
\text{(DISTR)} & b\ \{\ k\langle\vec{x}\rangle = c\ \}\ \{\ j\langle\vec{y}\rangle = d\ \} & = & b\ \{\ j\langle\vec{y}\rangle = d\ \}\ \{\ k\langle\vec{x}\rangle = c\ \{\ j\langle\vec{y}\rangle = d\ \}\ \} & \\
 & & & (\text{w.l.o.g., } k \neq j,\ j \notin \vec{x}, \text{ and } k, \vec{x} \notin \mathsf{FV}(d)) & \\
\text{(ETA)} & b\ \{\ k\langle\vec{x}\rangle = j\langle\vec{x}\rangle\ \} & = & b[j/k] & (\text{given } j \notin \vec{x}) \\
\text{(GC)} & k\langle\vec{x}\rangle\ \{\ j\langle\vec{y}\rangle = c\ \} & = & k\langle\vec{x}\rangle & (\text{given } j \notin \mathsf{FV}(k\langle\vec{x}\rangle))
\end{array}$$

The (JMP) rule effectively drives computation by allowing jumps to be performed to the closest bound continuation. In order to achieve that, the (DISTR) rule allows the reordering of continuations, which requires duplicating subterms due to the rigid syntax of the calculus. The (ETA) rule captures extensionality, and the (GC) rule states that the meaning of a term is unchanged by unused continuations in it. All the above naming restrictions in the rules above are used simply to state that they are to be defined in a capture-avoiding way in either direction.

We remark that the (JMP) rule may seem restrictive: it imposes the condition that the bound continuation k is not used in the parameters and then removes it on the right-hand side. This happens as these rules were given with a system of simple types in mind, and there is no simple type for $k\langle\vec{x}\rangle$ when $k \in \vec{x}$. The following rule however lifts these restrictions and is admissible, i.e., it is derivable from the above axioms:

$$\text{(RECJMP)} \qquad k\langle\vec{x}\rangle\ \{\ k\langle\vec{y}\rangle = c\ \} \quad = \quad c[\vec{x}/\vec{y}]\ \{\ k\langle\vec{y}\rangle = c\ \}$$

Notably the bound continuation is preserved after the jump.

The four axioms of the equational theory are also sound with regard to the two translations given above. Namely, if $b = c$, then $b^\circ = c^\circ$ in both λ-calculi [Thielecke 1997b], and $b^\bullet = c^\bullet$ in some variants of the π-calculus [Merro and Sangiorgi 2004]. The (DISTR) rule, for example, corresponds directly to a *sharpened replication axiom* in the π-calculus [Sangiorgi and Walker 2001].

2.3 Operational Semantics

Historically, the first formulation of an operational semantics for the CPS-calculus has been sketched in the PhD thesis of Thielecke [1997a]. He suggested the use of directed versions of the above (DISTR) and (JMP) axioms as computational steps, with the former bringing continuations to the right place for the latter to act upon. However, as (DISTR) also duplicates continuations, this may have undesirable consequences; for example, it cannot be strongly normalizing.

Later, Merro and Sangiorgi [2004] presented a generalization of this approach. They remark that every command may be represented by a jump in head position followed by a sequence of binds, and so, they propose a reduction semantics by using a reduction relation with a single global (i.e.,

non-compatible) rule that behaves as a contextual variant of the (RECJMP) rule, jumping to any bound continuation within a sequence of binds, preserving that sequences of binds after the jump:

$\boxed{b \longrightarrow b}$ *Head Reduction*

$$k_i\langle\vec{x}\rangle\ \{\ k_1\langle\overrightarrow{y_1}\rangle = c_1\ \}\ \cdots\ \{\ k_n\langle\overrightarrow{y_n}\rangle = c_n\ \} \quad\longrightarrow\quad c_i[\vec{x}/\overrightarrow{y_i}]\ \{\ k_1\langle\overrightarrow{y_1}\rangle = c_1\ \}\ \cdots\ \{\ k_n\langle\overrightarrow{y_n}\rangle = c_n\ \}$$
(given $1 \leq i \leq n$ and, w.l.o.g., $k_j \notin \mathsf{FV}(c_i)$ for all $j < i$)

The above relation, which we call *head reduction*, captures the intuition that a jump in head position is responsible for specifying the behavior of a term. By performing only the leftmost jumps, head reduction thus can simulate the evaluation semantics which was internalized by the CPS translation, acting *at a distance* [Accattoli and Kesner 2010] so as to preserve the syntactic structure.

A *head-redex* may be thus characterized by a jump at head position to a bound variable, given that arguments and parameters are of the same size; if there is a mismatch between the jump and its binding, a term is said to be *stuck*. Note that any closed term, unless it gets stuck, will necessarily loop, as reduction cannot introduce free variables and thus the jump at head position will always form a head redex (in a way, the term always knows how to continue). In a similar setting, this notion of reduction semantics also appears to have been independently described by Amadio and Régis-Gianas [2012] for an equivalent calculus (cf. Amadio [2017]).

2.4 Observational Semantics

The definition of an operational semantics naturally induces the idea of a *behavioral equivalence*: we consider two terms equivalent if we are free to replace one for the other in any context without possibly changing the result. This is particularly important for IRs, as we would like to optimize terms by replacing them by more efficient versions without changing the overall program behavior.

Since the only action that a term may perform is jumping, this is also the only way it may interact with its context. Informally, if a term has a jump to a free variable at head position, then an external observer may interact with it by binding such variable to a continuation. It is therefore possible to define a *convergence predicate* (in the terminology of Milner and Sangiorgi [1992]) which acts as a *barb* and states that a term may immediately perform an observable action:

$\boxed{b \downarrow_x}$ *Convergence*

$$(k\langle\vec{x}\rangle\ \{\ k_1\langle\overrightarrow{y_1}\rangle = c_1\ \}\ \cdots\ \{\ k_n\langle\overrightarrow{y_n}\rangle = c_n\ \})\ \downarrow_k \quad \text{(given } k \neq k_i \text{ for } 1 \leq i \leq n\text{)}$$

Convergence in a term is a notion similar to that of being a value in λ_n and λ_v, as in both cases this implies that there are no more redexes (neither under CBN nor CBV). We say that c *converges* to k if $c \downarrow_k$. Notice that, if a term converges to some k, then it is in *head-normal form*, and if it does not, the term is either a head-redex or is stuck. We also say that c *weakly converges* to k if $c \longrightarrow^* \downarrow_k$, i.e., if c converges to k after zero or more steps.

Several possible different notions of behavioral equivalence for the CPS-calculus were studied by Merro [2010], including variants of Morris' contextual equivalence, Milner and Sangiorgi's barbed congruence, and Abramsky's applicative bisimulation. Merro showed that all of these definitions actually coincide; this relation, which we will simply refer to as *bisimilarity*, may be defined among other ways as follows:

$\boxed{b \approx b}$ *Bisimilarity*

$$b \approx c \quad = \quad \forall C, \forall k,\ C[b] \longrightarrow^* \downarrow_k \iff C[c] \longrightarrow^* \downarrow_k$$

It follows by definition that if $b \approx c$ then these two terms cannot be *taken apart*[6] internally by any observer while using head reduction, and bisimilarity is indeed the largest relation closed under this property. Now, if we want to use head reduction as a semantics for the CPS-calculus, we would expect that the equational theory is also sound with respect to it: two terms deemed equal to each other by the equational axioms should not be able to be taken apart either. Fortunately, it is already known that this is the case, as head reduction is a sufficient and necessary condition to capture the computational content of a term.

Theorem 2.1 (Equational Soundness, Merro and Sangiorgi [2004]). *The equational theory is sound with regard to the observational equivalences. Thus, if* $b = c$, *then* $b \approx c$.

This theorem implies that, just as $\to_n$ and $\to_v$ work as *essential steps* [Accattoli et al. 2019], required for termination in the respective λ-calculi, so does head reduction for the CPS-calculus.

3 Operational Semantics, Revisited

We follow Plotkin's program by first taking into account the different possible ways of giving meaning to terms. While a calculus often has its meaning given through some reduction rules, the act of decomposing a term into a redex and context pair, performing the redex, and then recomposing them is costly from a computational point of view. So, programming languages (and by extension IR languages) will usually opt for a more direct approach: defining either an interpreter or an abstract machine. Just as the SECD machine for ISWIM interprets terms as if by β_v-reduction, so should a semantics meant for an IR match reductions in the CPS-calculus.

3.1 Machine Semantics

Though Merro and Sangiorgi's head reduction was the first operational semantics given for the CPS-calculus, it was not the first semantics given for the corresponding language. In his book, Appel [1991, ch. 3] defines a denotational semantics for his IR by writing an interpreter in the host language of his compiler. Later, Kennedy [2007] captures the idea behind such an interpreter in the form of an abstract machine semantics, which we take as the evaluator for our IR. Kennedy's machine is given in terms of a big-step relation of a term c underneath a *heap* ρ,[7] written as $\rho \vdash c \Downarrow$. Heaps are given as follows, adapted to the CPS-calculus' syntax:

$$\textit{Heaps} \quad \rho \ ::= \ \cdot \mid \rho,\ x = \underbrace{\langle \rho,\ \lambda \vec{x}.c \rangle}_{\textit{closure}}$$

That is, a heap works as an association list in which $\langle \sigma,\ \lambda \vec{y}.c \rangle$ represents a *closure* for continuations which captures, as a pair, another heap σ and a command c with parameters $\vec{y}$ (where we merely use the λ symbol to express this binding). The machine's rules are then defined:

$\boxed{\rho \vdash b \Downarrow}$ *Machine Semantics*

$$\frac{\rho,\ k = \langle \rho,\ \lambda \vec{y}.c \rangle \vdash b \Downarrow}{\rho \vdash b\ \{\ k\langle \vec{y} \rangle = c\ \} \Downarrow}\ (1) \qquad \frac{\rho(k) = \langle \sigma,\ \lambda \vec{y}.c \rangle \qquad \sigma,\ \vec{y} = \rho(\vec{x}) \vdash c \Downarrow}{\rho \vdash k\langle \vec{x} \rangle \Downarrow}\ (2) \qquad \frac{\rho(k) \text{ is undefined}}{\rho \vdash k\langle \vec{x} \rangle \Downarrow}\ (3)$$

These rules are adapted almost directly from Kennedy [2007], and defined in a straightforward way, working as the trace of an interpreter (going up the proof tree). Although the original interpretation makes a distinction between continuations and source-level functions, we generalize it for the case

[6] We say that terms b and c can be taken apart internally if they are semi-separable: there is a context C such that $C[b]$ halts and $C[c]$ diverges, so there is a program that can use these two terms in a way that it finds a difference between them.
[7] This corresponds to the usual notion of a machine environment, but we reserve this name for typing environments.

where continuations are first-class, similar to the machine later used by Bélanger and Appel [2017]. If we restrict the syntax of the CPS-calculus to keep continuations as second-class citizens, the original formulation is recovered.

If a term is a bind, by (1) then the bound continuation is added into the heap as a closure, capturing the current heap ρ, then computation proceeds with b. Otherwise, if a term is a jump, two things might happen: a jump is performed to a closure by (2) if it is defined, in which case its heap is extended to account for the arguments $\vec{x}$ being used to bind the parameters $\vec{y}$ that are defined[8] before proceeding, or computation halts by (3) because a jump is performed to a continuation which is not bound. When the heap is empty, we simply write $c \Downarrow$ instead of $\vdash c \Downarrow$, meaning that evaluation will eventually terminate, in a fresh heap, by jumping into a free variable.

The condition for termination is the same as convergence: the need to interact with the context in order to know how to proceed (after all, CPS terms do not return!). In fact, it is possible to notice a close relationship between this machine and head reduction: only the leftmost jumps are performed, and indeed the same number of times in each style. The following theorem formalizes this intuition, which means that the machine semantics characterizes weak convergence.

Theorem 3.1 (Machine Correctness). *The machine semantics is sound and complete with regard to head reduction. In other words, $c \Downarrow$ if and only if $c \to^* \downarrow_k$ for some k.*

Proof Sketch. For the "if" side (completeness), we show that if $b \to c$ and $c \Downarrow$, then $b \Downarrow$. Doing so requires a generalized lemma to apply substitution in a heap: we say that heap ρ corresponds to heap σ under a substitution of variables f at *time* n (that is, after performing n steps in the interpreter, which coincides with the height of the proof tree) if, for every variable x, the closure $\rho(x[f])$ corresponds to $\sigma(x)$ at n, and we say a closure $\langle \rho,\ \lambda\vec{x}.b\rangle$ corresponds to closure $\langle \sigma,\ \lambda\vec{y}.c\rangle$ at time n if, for all $m < n$, ρ', σ', f' and $\vec{z}$ such that ρ' corresponds to σ' under f' at m and $\rho,\ \vec{x} = \rho'(\vec{z}[f']) \vdash b \Downarrow$ with the proof tree having height m, we have that $\sigma,\ \vec{y} = \sigma'(\vec{z}) \vdash c \Downarrow$ with the proof tree having height m. We then prove, by induction over the height of the proof tree, that if ρ corresponds to σ under f and $\rho \vdash c[f] \Downarrow$, then $\sigma \vdash c \Downarrow$, which is enough to show that a head-redex is valid. For the "only if" side (soundness), we give an interpretation $[\![-]\!]$ turning heaps into static contexts, where we define that $[\![\cdot]\!] = [-]$ and $[\![\rho,\ k = \langle \sigma,\ \lambda\vec{y}.c\rangle]\!] = [\![\rho]\!][[-]\ \{\ k\langle\vec{y}\rangle = [\![\sigma]\!][c]\ \}]$, and we prove, by induction on the proof tree, that $\rho \vdash c \Downarrow$ implies that $[\![\rho]\!][c] \to^* \downarrow_k$ for some k by using bisimilarity between heaps at each step of the tree. □

A simple consequence of this result is that the notion of behavioral equivalence between b and c in the IR, namely that $C[b] \Downarrow$ if and only if $C[c] \Downarrow$ for any C, actually coincides with bisimilarity: two terms can be taken apart in the IR if and only if they can be taken apart in the CPS-calculus. We note that an alternative definition for a behavioral equivalence in the IR using heaps instead of contexts, given as $\rho \vdash b \Downarrow$ if and only if $\rho \vdash c \Downarrow$ for any ρ, will still coincide with bisimilarity.

Theorem 3.2 (Machine Equivalence Characterization). *Bisimilarity for terms b and c can be characterized by the definitions of behavioral equivalences in the IR. I.e., $b \approx c$ if and only if:*

(1) $\forall C, C[b] \Downarrow \iff C[c] \Downarrow$
(2) $\forall \rho, \rho \vdash b \Downarrow \iff \rho \vdash c \Downarrow$

Proof. For (1), by Theorem 3.1, it is enough to show that if $C[b] \downarrow_k$ and $C[c] \downarrow_j$ for $k \neq j$, there is a discriminating context where $b \Downarrow$ but not $c \Downarrow$, which is a contradiction. For (2), we use the Context Lemma given by Merro [2010] that says we only need to consider static contexts in a contextual equivalence, and use the heap interpretation from Theorem 3.1 to show the correspondence. □

[8] Note that for any $x_i \in \vec{x}$, if $\rho(x_i)$ is undefined, then the corresponding $y_i \in \vec{y}$ is also not defined on the extended heap.

Again, we recall that CPS is not necessarily useful only for compiling languages with control effects [Kennedy 2007]. Although the notion of bisimilarity would be different if we consider restrictions to prevent such effects, the result from Theorem 3.2 would still hold.

3.2 Reduction Semantics

The CPS-calculus, as studied by Merro [2010], is actually deterministic: each term has, at most, a single head-redex. This is somewhat restrictive, as the equivalence closure for this relation equates too few terms under head-normal form. We proceed to look for a larger reduction relation satisfying some desirable properties, but that remains sound with regard to the equational theory.

In a similar manner to how Plotkin demonstrated that the CBV reduction strategy was not as order-sensitive as previously thought, we generalize the notion of head reduction $\to$ (Section 2.3) by allowing jumps to be performed anywhere in a term. The derived notion of *jump reduction* is taken as the compatible closure of the following rule:

$\boxed{b \rightsquigarrow_j b}$ *Jump Reduction*

$$C[k\langle\vec{x}\rangle]\ \{\ k\langle\vec{y}\rangle = c\ \} \quad \rightsquigarrow_j \quad C[c[\vec{x}/\vec{y}]]\ \{\ k\langle\vec{y}\rangle = c\ \} \qquad \text{(w.l.o.g., } k \notin \mathsf{dom}(C)\text{)}$$

It follows immediately that $\to \subset \rightsquigarrow_j$ since a head-redex can be seen as a *jump-redex* taking place strictly in static contexts. A term is therefore in *jump-normal form* if it contains no jump to any defined continuation anywhere inside of it. While this allows for a finer notion of normal form, which is able to identify strictly more terms then by head reduction alone, unfortunately performing jump reductions can quickly result in a blowup in the term's size. Consider a simple sizing function $|-|$ which counts the number of jumps in a term: we can check that if $b \rightsquigarrow_j c$, then $|b| \leq |c| < 2|b|$. Even though a term can almost double in size at the worst case, it will never decrease.

In order to better deal with ever-growing terms, we also introduce a *garbage collection* step, which removes continuations which are no longer being used, as the compatible closure of a slight generalization of the (GC) axiom (Section 2.2):

$\boxed{b \rightsquigarrow_g b}$ *Garbage Collection Reduction*

$$b\ \{\ k\langle\vec{y}\rangle = c\ \} \quad \rightsquigarrow_g \quad b \qquad \text{(given } k \notin \mathsf{FV}(b)\text{)}$$

We now derive our notion of *full reduction*, which we denote by $\rightsquigarrow$, by taking the union of $\rightsquigarrow_j$ and $\rightsquigarrow_g$. This new notion of reduction is capable of equating more terms under a *full-normal form*, but, nevertheless, it is still sound, and conservative: additional reductions cannot be observed.

$\boxed{b \rightsquigarrow b}$ *Full Reduction*

$$\rightsquigarrow \quad = \quad \rightsquigarrow_j \cup \rightsquigarrow_g$$

Lemma 3.3 (Reduction Soundness). *Full reduction is sound with regard to the equational theory. Thus, if* $b \rightsquigarrow c$, *then* $b = c$. *As a consequence,* $b \approx c$ *as well.*

Theorem 3.4 (Convergence Characterization). *The relations* $\to^*\downarrow_k$ *and* $\rightsquigarrow^*\downarrow_k$ *coincide. As a consequence, bisimilarity for terms* b *and* c *can be characterized by the contextual equivalence derived from full reduction. I.e.,* $b \approx c$ *if and only if* $\forall C, \forall k,\ C[b] \rightsquigarrow^*\downarrow_k \iff C[c] \rightsquigarrow^*\downarrow_k$.

Proof. Follows directly from Lemma 3.3 and the definition of bisimilarity, as $\to^* \subset \rightsquigarrow^* \subset \approx$. □

At first glance, this notion of reduction may look like an obvious generalization of head reduction. Remarkably it has, however, appeared before in the literature in multiple seemingly unrelated

places. First, it appears exactly as *optimization rules* for IRs, where jump and garbage collection steps represent, respectively, *function inlining* and *dead code elimination* [Appel and Jim 1997; Kennedy 2007]. In fact, function inlining has already been suggested as a semantics for an IR by Appel and Jim [1997], but, to the best of our knowledge, this had not been explored before. This same exact relation has also appeared as a proof method for the π-calculus [Honda and Laurent 2010; Yoshida et al. 2004], where it has been shown to be justified by the cut-elimination rules on linear proof nets. Under the same considerations, it has also appeared in a more granular variant as a reduction relation for explicit substitution calculi inspired by linear logic [Accattoli et al. 2014; Accattoli and Kesner 2010] and for the call-by-need λ-calculus [Ariola et al. 1995].

3.3 Shrinking Reductions

Even though our full reduction is sound with regard to the equational theory of the calculus, it is still not complete. Ideally, we would like to find a notion of reduction with good properties and whose congruence closure approximates the equational theory up to some decidable structural equivalence, so that we can simply orient the axioms in order to have the reduction rules. Following the idea that we are taking compiler optimization rules as a reduction semantics, we turn our attention to the notion of *shrinking reductions*.

A shrinking reduction [Appel and Jim 1997] is a kind of optimization which necessarily decreases the size of a term, and thus may be applied exhaustively. Examples of shrinking reductions include dead code elimination, linear function inlining (where a function is called only once), tail call elimination and contification [Kennedy 2007], for which there are near-linear time algorithms by representing terms as a graph [Benton et al. 2004]. Though we proceed with our development by using the garbage collection step alone, we note that it would be possible to take other notions of shrinking reduction instead of $\rightsquigarrow_g$, using some abstract $\rightsquigarrow_s$. We consider a notion of a shrinking reduction as some reduction relation that satisfies the following *tidying requirements*, which are needed for preserving our results (cf. Appel and Jim [1997, appendix]).

Definition 3.5 (Tidying Requirements). A notion of reduction $\rightsquigarrow_s$ satisfies the tidying requirements if it contains the following properties:

(1) Shrinking: it should be strongly normalizing, strictly decreasing in some notion of size.
(2) Soundness: it should be included in the equational theory, so $b \rightsquigarrow_s c$ should imply $b = c$.
(3) Confluence: it should be confluent, so that if $b \rightsquigarrow_s^* c_1$ and $b \rightsquigarrow_s^* c_2$, then there is some d such that both $c_1 \rightsquigarrow_s^* d$ and $c_2 \rightsquigarrow_s^* d$.
(4) Commutation: it should commute with jumping, so that if $b \rightsquigarrow_s^* c_1$ and $b \rightsquigarrow_j^* c_2$, then there is some d such that both $c_1 \rightsquigarrow_j^* d$ and $c_2 \rightsquigarrow_s^* d$.
(5) Reordering: it should be able to be reordered with regard to jumping, so $b \rightsquigarrow_s \rightsquigarrow_j^+ c$ should imply $b \rightsquigarrow_j^+ \rightsquigarrow_s^* c$. As a consequence, shrinking steps can't create jumps.

From these conditions, it is possible to derive that the union of a shrinking reduction with jumping preserves the desirable properties for a reduction relation. This is, of course, the case of garbage collection, which is indeed a shrinking reduction.

Theorem 3.6 (Tidying Preservation). *Let $\rightsquigarrow_s$ be a shrinking reduction, and let $\mathcal{R} = \rightsquigarrow_j \cup \rightsquigarrow_s$. Then confluence, factorization, and strong normalization of $\rightsquigarrow_j$ imply the respective properties for $\mathcal{R}$.*

Corollary 3.7 (Garbage Collection). *The relation $\rightsquigarrow_g$ is a shrinking reduction, and thus $\rightsquigarrow$ preserves confluence, factorization, and strong normalization for $\rightsquigarrow_j$.*

As described by Appel and Jim [1997], the requirements for computation and for optimization are not quite the same. For example, confluence is not necessary for optimization, but is still a

useful property as this allows for the design of different optimization algorithms with regard to optimization order without having to reason about the resulting terms, solving a *phase ordering* problem. The tidying requirements above thus allow us to consider both computation rules (arbitrary function inlining) and practical optimization rules within the same framework without needing to deal with extra technical details, while still preserving the metatheory of the CPS-calculus.

Still, we remark that it is unclear what constitutes a "complete" set of shrinking reductions, and that there could be different conflicting notions of *shrink-normal form*. While dead code elimination satisfies the tidying requirements above, adding tail call elimination to it, which is simply an oriented version of the (ETA) rule, breaks confluence. For example, the term $\mathsf{x}\langle \mathsf{k},\ \mathsf{j}\rangle\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{j}\langle\vec{\mathsf{y}}\rangle\ \}\ \{\ \mathsf{j}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}$ may perform tail call elimination leading to $\mathsf{x}\langle \mathsf{j},\ \mathsf{j}\rangle\ \{\ \mathsf{j}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}$, while performing a jump leads to $\mathsf{x}\langle \mathsf{k},\ \mathsf{j}\rangle\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{j}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}$, both of which might be in normal form. To join those two terms together, a *contraction* step is needed, as a directed, contextual variant of the following rule:

$$\text{(CONTR)}\qquad \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{j}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \} \quad = \quad \mathsf{b}[\mathsf{k}/\mathsf{j}]\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \} \qquad (\text{given } \mathsf{j} \notin \mathsf{FV}(\mathsf{c}) \text{ and } \mathsf{k}, \mathsf{j} \in \mathsf{FV}(\mathsf{b}))$$

This rule, akin to identical code folding optimization, had been taken as an additional axiom in previous works [Führmann and Thielecke 2004], but is in fact admissible and may be derived in the equational theory. Additionally, the converse of a directed contraction step, *specialization*, could also be considered (such as done by Accattoli and Kesner [2010]), though these two notions are incompatible with each other, since by undoing each other they wouldn't terminate. We thus leave the investigation of larger or more interesting notions of shrinking reductions for future work.

3.4 Structural Equivalence

We also remark that it might be useful to consider additional structural rules for the CPS-calculus, similar to the structural congruence used in the π-calculus and in some explicit substitution calculi [Accattoli and Kesner 2010]. As an example of possible choices, Thielecke [1997a] gives a decomposition of the (DISTR) rule into a couple of *floating* rules for the linear variant of the CPS-calculus as follows:

$$\text{(FLOAT-L)}\qquad \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{j}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \} \quad = \quad \mathsf{b}\ \{\ \mathsf{j}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \} \qquad (\text{given } \mathsf{k} \neq \mathsf{j} \text{ and } \mathsf{j} \notin \mathsf{FV}(\mathsf{c}))$$

$$\text{(FLOAT-R)}\qquad \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{j}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \} \quad = \quad \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \{\ \mathsf{j}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \}\ \} \qquad (\text{given } \mathsf{k} \neq \mathsf{j},\ \mathsf{j} \notin \mathsf{FV}(\mathsf{b}),\ \text{and } \vec{\mathsf{y}} \notin \mathsf{FV}(\mathsf{d}))$$

The rules are named after which continuation, b (on the left) or c (on the right), has a reference for the continuation j. Although these rules are taken as axioms in the linear CPS-calculus, they are admissible in the variant used in this work. Notice that (FLOAT-L) cannot possibly lead to the duplication of any redex through jumping, but (FLOAT-R) can. It still does so in a bounded way, as we will prove in Section 7. One could, then, work with full reduction modulo floating if so desired.

3.5 Summary so Far

Following Plotkin's steps, we have compared the machine semantics for the IR (which is Kennedy's refinement of Appel's interpreter) to the CPS-calculus' equational theory, showing that it is indeed equivalent to a notion of reduction through essential steps (Section 3.1). We extended this notion of reduction by allowing extra steps which cannot be observed (Section 3.2) but that nevertheless are sound within the theory, based on compiler optimization rules. As this reduction is still by no means complete, we entertain the idea of extending it in a way that preserves desired properties in order to accommodate for more optimizations (Section 3.3) and structural rules (Section 3.4). Now,

we continue[9] this line of work, as Plotkin did, by checking that this reduction semantics is enough to show that we can correctly compile programs into the CPS-calculus.

4 Computational Adequacy

The purpose of an IR is, of course, to serve as a compilation target. Appel's IR has been battle-tested in production compilers for a long time, and is clearly suited for the job. But, as the last section demonstrated that computation on his IR matches that of the CPS-calculus, we now have evidence that we can use the CPS-calculus as a tool for reasoning about code within a compiler. We proceed to do so by using the CPS-calculus to demonstrate the unsurprising but still important result that Appel's IR is indeed an adequate compilation target for both the λ-calculi.

4.1 CPS Translations

Among his other contributions, Plotkin [1975] introduces two CPS translations of the λ-calculi into each other and shows that they capture the intended meaning of a program. Namely, the CBN translation for the λ-calculus allows its execution to be simulated in the λ_v-calculus, and, similarly, the CBV translation for the λ_v-calculus allows its execution to be simulated in the λ-calculus. Even though these two translations are not the most efficient ones, as they introduce some administrative redexes, we chose them for both their simplicity and their historical importance.

The CPS translations $[\![-]\!]_{\mathsf{N}}$ and $[\![-]\!]_{\mathsf{V}}$ into the CPS-calculus, respectively for CBN and CBV, were already given by Thielecke [1997a] and are defined as follows:

$\boxed{[\![\mathbf{e}]\!]_{\mathsf{N}} = \mathbf{b}}$ *Plotkin's call-by-name translation*

$$
\begin{aligned}
[\![\mathbf{x}]\!]_{\mathsf{N}} &= \mathbf{x}\langle \mathbf{k} \rangle \\
[\![\lambda \mathbf{x}.\mathbf{e}]\!]_{\mathsf{N}} &= \mathbf{k}\langle \mathbf{v} \rangle \ \{\ \mathbf{v}\langle \mathbf{x},\ \mathbf{k} \rangle = [\![\mathbf{e}]\!]_{\mathsf{N}}\ \} \\
[\![\mathbf{f}\ \mathbf{e}]\!]_{\mathsf{N}} &= [\![\mathbf{f}]\!]_{\mathsf{N}}\ \{\ \mathbf{k}\langle \mathbf{f} \rangle = \mathbf{c}\ \{\ \mathbf{v}\langle \mathbf{k} \rangle = [\![\mathbf{e}]\!]_{\mathsf{N}}\ \}\ \}
\end{aligned}
$$

$\boxed{[\![\mathbf{e}]\!]_{\mathsf{V}} = \mathbf{b}}$ *Plotkin's call-by-value translation*

$$
\begin{aligned}
[\![\mathbf{x}]\!]_{\mathsf{V}} &= \mathbf{k}\langle \mathbf{x} \rangle \\
[\![\lambda \mathbf{x}.\mathbf{e}]\!]_{\mathsf{V}} &= \mathbf{k}\langle \mathbf{v} \rangle \ \{\ \mathbf{v}\langle \mathbf{x},\ \mathbf{k} \rangle = [\![\mathbf{e}]\!]_{\mathsf{V}}\ \} \\
[\![\mathbf{f}\ \mathbf{e}]\!]_{\mathsf{V}} &= [\![\mathbf{f}]\!]_{\mathsf{V}}\ \{\ \mathbf{k}\langle \mathbf{f} \rangle = [\![\mathbf{e}]\!]_{\mathsf{V}}\ \{\ \mathbf{k}\langle \mathbf{v} \rangle = \mathbf{c}\ \}\ \}
\end{aligned}
$$

In both translations, we take $\mathbf{c} = \mathbf{f}\langle \mathbf{v},\ \mathbf{k} \rangle$.[10] Variables that do not appear in the source term are considered fresh, namely $\mathbf{f}$, $\mathbf{v}$ (which are immediately bound and thus the choice of variable name does not matter when considering terms up to α-convertibility), and $\mathbf{k}$. Both translations expect $\mathbf{k}$ to be free, and, in fact, a single $\mathbf{k}$ may be used [Danvy et al. 1999], as both translations bind it in the context for the translation of subterms. We can recover Plotkin's original translations (up to β-equality, as his translations did not enforce that values are named) as $\lambda \mathbf{k}.[\![-]\!]^{\circ}_{\mathsf{N}}$ and $\lambda \mathbf{k}.[\![-]\!]^{\circ}_{\mathsf{V}}$. We remark the duality that arises between the two translations [Levy 2001; Wadler 2003].

4.2 Simulation

We follow the method of Yoshida et al. [2004] for proving adequacy, where they take the same reduction relation as full reduction but in the π-calculus and show that adequacy for the CBN translation holds. They do so by first proving a *simulation* lemma: if any reduction step is taken in the source calculi, the respective translation should be able to take some reduction steps in order to match it. The CBV translation, however, is a lot trickier.

Lemma 4.1 (Simulation). *Reductions on the target calculi may be simulated on the CPS-calculus.*

(1) If $\mathbf{e} \to_{\mathbf{n}} \mathbf{f}$, *then* $[\![\mathbf{e}]\!]_{\mathsf{N}} \to^{+} \rightsquigarrow^{*} [\![\mathbf{f}]\!]_{\mathsf{N}}$.

(2) Let $\mathcal{R}$ *be the least reduction relation closed under* $\rightsquigarrow$, *specialization and floating. If* $\mathbf{e} \to_{\mathbf{v}} \mathbf{f}$, *then* $[\![\mathbf{e}]\!]_{\mathsf{V}} \to^{+}_{\mathsf{e}} \mathcal{R}^{*} [\![\mathbf{f}]\!]_{\mathsf{V}}$.

[9] Pun intended. We apologize for this.

[10] Interestingly, in both contexts, this term becomes $1\langle 0,\ 2 \rangle$ in a de Bruijn setting such as used in our formalization.

Proof. By induction and case analysis. Performing substitution requires an extra induction step on the number of free occurrences of the parameter within the redex. □

In the above, we take $\rightarrow_e$ as a small generalization of head reduction, whose formalization we omit, that may be applied in any context that syntactically guarantees that the hole will be moved into head position, thus necessarily decreasing the necessary number of steps into head-normal form. So, we know that if a computation can proceed in the source, it can also proceed in the target. All we have to show now is the complementary property, that if computation has halted in the source, it will also halt in the target.

Lemma 4.2 (Termination). *The translations preserve termination. Namely, the following holds:*

(1) If there is no $\mathbf{f}$ *such that* $\mathbf{e} \rightarrow_n \mathbf{f}$*, then* $[\![\mathbf{e}]\!]_N$ *converges.*
(2) If there is no $\mathbf{f}$ *such that* $\mathbf{e} \rightarrow_v \mathbf{f}$*, then* $[\![\mathbf{e}]\!]_V$ *weakly converges.*
(3) If $\mathbf{e}$ *is in* β*-normal form, then* $[\![\mathbf{e}]\!]_N$ *is in full-normal form.*
(4) If $\mathbf{e}$ *is in* β_v*-normal form, then* $[\![\mathbf{e}]\!]_V$ *is strongly normalizing.*

Proof. Statements (1) and (2) follow by induction on the structure of the term. Statement (3) follows by induction, while noticing that a target term may only contain jumps to variables that were free in the source term or to the distinguished continuation k. Statement (4) follows as (3), by noticing that the only existing redexes are linear jumps, which are shrinking reductions. □

We can observe that full reduction is not enough to exactly simulate β_v-reduction for CBV. In order to see why, recall that a β-redex and a β_v-redex both replace each occurrence of the parameter by a copy of the argument, and in both cases, this happens through a continuation x which is only bound in the context for $\mathbf{e}$'s translation inside a redex. But, in $[\![(\lambda \mathsf{x}.\mathbf{e})\ \mathbf{v}]\!]_V$, the argument needs to be copied through specialization and moved to the right place through floating as x is used as an argument in the jump, as $[\![\mathbf{x}]\!]_V = \mathsf{k}\langle \mathsf{x} \rangle$. On the other hand, in the CBN version this can be done through jumping alone, as $[\![\mathbf{x}]\!]_N = \mathsf{x}\langle \mathsf{k} \rangle$. This issue is indeed not surprising, as the CBN translation can be seem as a special case of the CBV translation, since the CBN one can be decomposed into *thunking* in the source language, followed by the CBV one (sometimes regarded as the de-facto CPS translation) and then by the simplification of administrative redexes [Hatcliff and Danvy 1997]. However, this mismatch in the reduction relation does not prevent adequacy of CBV. We conjecture that there exists a notion of shrinking reduction that may be used to simplify this result.

4.3 Adequacy

The CPS translations may be seen as a form of denotational semantics: we can give meaning to terms in the λ-calculi by demonstrating how they translate into the CPS-calculus. For such interpretation to be considered correct, then all observably distinct programs should have distinct denotations. Informally, this can be seen as stating that we cannot possibly compile two programs distinct in the source language's semantics into the same CPS term.

The only observable action that terms may have in the λ-calculi is termination. We follow the usual tradition of identifying programs with closed terms, but note that these notions naturally generalize to open terms as shown by Abramsky [1990]. In the following, we define $\mathbf{e} \Downarrow_N$ to mean that there is a $\mathbf{v}$ such that $\mathbf{e} \rightarrow_n^* \mathbf{v}$ and $\mathbf{e} \approx_N \mathbf{f}$ if for all contexts $\mathbf{C}$ such that $\mathbf{C}[\mathbf{e}]$ and $\mathbf{C}[\mathbf{f}]$ are closed, $\mathbf{C}[\mathbf{e}] \Downarrow_N$ if and only if $\mathbf{C}[\mathbf{f}] \Downarrow_N$, with corresponding $\mathbf{e} \Downarrow_V$ and $\mathbf{e} \approx_V \mathbf{f}$ for CBV. We can now conclude that Appel's IR is an adequate compilation target for the λ-calculi.

Theorem 4.3 (Computational Adequacy). *The CPS translations are computationally adequate. Let* $\mathbf{e}$ *be a closed term, then* $\mathbf{e} \Downarrow_N$ *if and only if* $[\![\mathbf{e}]\!]_N \Downarrow$*, and* $\mathbf{e} \Downarrow_V$ *if and only if* $[\![\mathbf{e}]\!]_V \Downarrow$*.*

Proof. Follows from Lemmas 4.1 and 4.2. The "only if" sides is similar to the proof in Yoshida et al. [2004]. For CBN, the "if" side follows by induction on the reduction path to head-normal form, by reordering steps using Theorem 6.1. For CBV, as reduction cannot be simply reordered due to floating, the "if" case follows by induction on the number of steps to head-normal form instead, as $\to_e$ decreases this amount and floating doesn't change it. □

As a consequence, we have that the CPS translations are sound as a denotational semantics.

Corollary 4.4 (Denotational Soundness). *Both CPS translations map distinct terms into distinct results. Let* $\mathbf{e}$ *and* $\mathbf{f}$ *be closed terms. Then* $[\![\mathbf{e}]\!]_N \approx [\![\mathbf{f}]\!]_N$ *implies* $\mathbf{e} \approx_N \mathbf{f}$*, and* $[\![\mathbf{e}]\!]_V \approx [\![\mathbf{f}]\!]_V$ *implies* $\mathbf{e} \approx_V \mathbf{f}$.

The converse of Corollary 4.4, however, is not true: for example, $\mathbf{e} \approx_N \mathbf{f}$ is not enough to have $[\![\mathbf{e}]\!]_N \approx [\![\mathbf{f}]\!]_N$. Such a property, called *full abstraction* [Ahmed and Blume 2011], would mean that the translation not only reflects but also preserves equalities. This however does not hold here as the CPS-calculus may distinguish more terms than the λ-calculi due to control effects which are not present in the source languages as they were presented. In a variant of the CPS-calculus that does not allow for control effects, such as by using the restrictions from Kennedy's IR, it would be possible to derive full abstraction by using a back-to-DS translation [Ahmed and Blume 2011; Danvy and Lawall 1996].

Even so, the relationship between evaluation in Appel's IR and in the CPS-calculus given by Theorem 3.1, along with the adequacy result of Corollary 4.4, demonstrates that the CPS-calculus is suitable for reasoning about optimizations within a compiler. If an optimization can be shown to change terms only up to bisimilarity, then it follows that it does not change the observable behavior from the source program as well. Thus, to prove an optimization correct one can freely benefit from the equational theory, similar to how it has been done for the IR given by Maurer et al. [2017], as well as from bisimulation proof methods such as those presented by Merro [2010].

4.4 Back to Compilers

In the previous section, we extended the work of Merro [2010] by showing more characterizations of the bisimilarity relation, using either the machine semantics or full reduction (Theorems 3.2 and 3.4, respectively). We can then leverage the results above to reason about optimizations.

One of the key motivations for designing a theoretical foundation for an IR should be, of course, its practical use within a compiler. To illustrate this, we now proceed to give a simple proof that a common optimization technique, *contification*, is sound. As discussed, Kennedy's IR departs from Appel's by the use of some syntactic restrictions (which are respected by the above CPS translations) such as treating continuations as second-class citizens, which is enough to rule out control effects. As there is a syntactic distinction between functions and continuations, the contification optimization is then used to check for runtime functions which may be turned into continuations, thus avoiding allocation and turning function calls into jumps.

The contification transformation, which we adapt directly from Kennedy [2007], may be described as the compatible closure of the following rewriting rule, where H is a multi-hole context (whose formal definition we omit for brevity):

$\boxed{\mathsf{cont}(\mathsf{b}) = \mathsf{b}}$ *Contification*

$$\mathsf{cont}(\mathsf{C}[\mathsf{H}[\mathsf{f}\langle \mathsf{x}_1, \mathsf{j}\rangle, ..., \mathsf{f}\langle \mathsf{x}_n, \mathsf{j}\rangle]]\ \{\ \mathsf{f}\langle \mathsf{y},\ \mathsf{k}\rangle = \mathsf{c}\ \}) \quad = \quad \mathsf{C}[\mathsf{H}[\mathsf{h}\langle \mathsf{x}_1\rangle, ..., \mathsf{h}\langle \mathsf{x}_n\rangle]\ \{\ \mathsf{h}\langle \mathsf{y}\rangle = \mathsf{c}[\mathsf{j}/\mathsf{k}]\ \}]$$
$$(\mathsf{f}, \mathsf{h} \notin \mathsf{FV}(\mathsf{C}) \cup \mathsf{FV}(\mathsf{H}) \cup \{\mathsf{j}, \mathsf{x}_1, ..., \mathsf{x}_n\}, \text{ and } \mathsf{H} \text{ minimal})$$

We note that every call to the function f is given the same continuation j to which it should return, and as such, we can actually turn f into a fresh continuation h as it is always supposed to return to

the same point. Consequently, in the result of contification, every call to f becomes a jump to its new continuation form, which is now defined inside of C to ensure that the continuation j is within scope for c. In order to prove that this transformation is sound, we use the equational theory as a tool to derive the desired bisimilarity between terms:

Theorem 4.5 (Contification). *The contification transformation is sound. Thus, for every* c*, it holds that* $c \approx \mathtt{cont}(c)$.

Proof. By Theorem 2.1, we show that, if $\mathtt{cont}(b) = c$, then $b = c$. By applying the floating rules on the left-hand side, which are admissible in the equational theory, we need to show that $C[H[f\langle x_1, j\rangle, \ldots, f\langle x_n, j\rangle]\ \{\ f\langle y,\ k\rangle = c\ \}]$ is equivalent to $C[H[h\langle x_1\rangle, \ldots, h\langle x_n\rangle]\ \{\ h\langle y\rangle = c[j/k]\ \}]$. By induction on the number of holes in H, applying the jump in each hole, and then applying a garbage collection step, both steps which are sound by Lemma 3.3, then both sides become $C[H[c[x_1/y, j/k], \ldots, c[x_n/y, j/k]]]$. □

We can conclude that $\mathbf{e} \Downarrow_N$ if and only if $\mathtt{cont}^*([\![\mathbf{e}]\!]_N) \Downarrow$ and also accordingly for the CBV, which shows that contification cannot change the meaning of a source program for either λ_n and λ_v.

5 Confluence

The full reduction relation introduced in Section 3 is a sound extension to head reduction, and it is no longer deterministic. This allows for additional steps to be performed, at the cost of introducing the notion of evaluation order in the CPS-calculus. Though this is distinct from the evaluation order captured by the CPS translation, which can be simulated through head reduction alone, having different evaluation orders imposes similar issues as they do in a programming language. Can we be sure that performing computation in two different ways will lead to the same result? In order to do so, we have to show that full reduction is confluent.

Confluence guarantees that no matter which reduction order we pick, or how many reduction steps we take in each direction, two reduction sequences can always be joined back together. For proving confluence, if some relation $\mathcal{R}$ has the diamond property, i.e., if $b\ \mathcal{R}\ c_1$ and $b\ \mathcal{R}\ c_2$ imply that there is a d such that $c_1\ \mathcal{R}\ d$ and $c_2\ \mathcal{R}\ d$, it is well known that $\mathcal{R}$ is confluent. However, as is the case for β-reduction in the λ-calculus, this does not hold for full reduction in the CPS-calculus. Consider the following example:

$$
\begin{array}{ccccc}
 & & (1)\quad k\langle\rangle\ \{\ k\langle\rangle = j\langle\rangle\ \}\ \{\ j\langle\rangle = c\ \} & & \\
 & \swarrow & & \searrow & \\
(2)\quad k\langle\rangle\ \{\ k\langle\rangle = c\ \}\ \{\ j\langle\rangle = c\ \} & & & & j\langle\rangle\ \{\ k\langle\rangle = j\langle\rangle\ \}\ \{\ j\langle\rangle = c\ \}\quad (3) \\
 & \searrow & & & \\
 & & (4)\quad c\ \{\ k\langle\rangle = c\ \}\ \{\ j\langle\rangle = c\ \} & &
\end{array}
$$

Term (3) can reduce to (4), but it requires two jumps to do so, and they can be done in any order, so there are two possible reduction paths for it. Things get still more complicated when considering garbage collection steps. But, as confluence for full reduction follows from confluence for jump reduction alone by Corollary 3.7, we only need to specify how to join two distinct sequence of jump reductions back together.

5.1 Residuals

The usual trick for proving confluence is by using a notion of *parallel reduction*: if we have some relation $\mathcal{R}$ such that $\rightsquigarrow_j \subset \mathcal{R}^? \subset \rightsquigarrow_j^*$, then, as $\mathcal{R}^* = \rightsquigarrow_j^*$, the diamond property of $\mathcal{R}^?$ implies confluence for $\rightsquigarrow_j$. In the λ-calculus, the notion of parallel β-reduction has a simple and elegant

inductive characterization as given by Takahashi [1989], but, as jump reduction works at a distance and needs to quantify over contexts, a similar approach does not work for it. Instead, we use an older proof technique, by building a theory of *residuals* [Barendregt 1981], similar to how it's been done for explicit substitution by Accattoli et al. [2014].

Consider some reduction $b \rightsquigarrow_j^+ c$. If we take note of some set of redexes that exist in b, and keep track of what happens to them while performing the reduction (either by erasure or duplication), we will now have a corresponding set of redexes in c, which we call the residuals. In order to do so, we give a syntactic representation for sets of redexes as follows:

$$\textit{Redexes} \quad \underline{r} \ ::= \ x\langle\vec{x}\rangle \ \mid \ \underline{x\langle\vec{x}\rangle} \ \mid \ \underline{r}\ \{\ x\langle\vec{x}\rangle = \underline{r}\ \}$$

A set of redexes $\underline{r}$ for a term b is given by taking the same structure as b while *marking* the jumps that belong to the set by underlining them. This leads to a natural inclusion with b itself acting as an empty set of redexes (since it cannot contain any marks). In order to track what happens to redexes in a term, we additionally give a list of mark definitions:

$$\textit{Definitions} \quad \delta \ ::= \ \cdot \ \mid \ \delta,\ x = \lambda\vec{x}.\underline{r}$$

We can now define the residuals relation as follows:

$\boxed{\delta \vdash \underline{r}\backslash\underline{r} = \underline{r}}$ *Residuals Relation*

$$\frac{}{\delta \vdash k\langle\vec{x}\rangle \ \backslash \ k\langle\vec{x}\rangle \ = \ k\langle\vec{x}\rangle}\ (1) \qquad \frac{}{\delta \vdash \underline{k\langle\vec{x}\rangle} \ \backslash \ k\langle\vec{x}\rangle \ = \ \underline{k\langle\vec{x}\rangle}}\ (2)$$

$$\frac{\delta(k) = \lambda\vec{y}.\underline{r}}{\delta \vdash k\langle\vec{x}\rangle \ \backslash \ \underline{k\langle\vec{x}\rangle} \ = \ \underline{r}[\vec{x}/\vec{y}]}\ (3) \qquad \frac{\delta(k) = \lambda\vec{y}.\underline{r}}{\delta \vdash \underline{k\langle\vec{x}\rangle} \ \backslash \ \underline{k\langle\vec{x}\rangle} \ = \ \underline{r}[\vec{x}/\vec{y}]}\ (4)$$

$$\frac{\delta \vdash \underline{s_1}\backslash\underline{s_2} = \underline{s_3} \qquad \delta,\ k = \lambda\vec{y}.\underline{s_3} \vdash \underline{r_1}\backslash\underline{r_2} = \underline{r_3}}{\delta \vdash \underline{r_1}\ \{\ k\langle\vec{y}\rangle = \underline{s_1}\ \} \ \backslash \ \underline{r_2}\ \{\ k\langle\vec{y}\rangle = \underline{s_2}\ \} \ = \ \underline{r_3}\ \{\ k\langle\vec{y}\rangle = \underline{s_3}\ \}}\ (5)$$

Assume that $\delta \vdash \underline{r}\backslash\underline{s} = \underline{t}$. We say that redexes $\underline{t}$ are the residuals of $\underline{r}$ after $\underline{s}$, meaning that $\underline{r}$ reduces to $\underline{t}$ by performing all the jumps marked by $\underline{s}$. We note that the above is a partial function, and it is only defined if $\underline{r}$ and $\underline{s}$ are *compatible* with each other, i.e., they represent redexes for the same underlying term. As each marked jump in $\underline{s}$ has to be performed, the residuals are also not defined unless $\underline{s}$ is *well-marked*, meaning that all the marked redexes are actually jumps to a bound continuation and with the right arity. In the following, we write $\underline{r}\backslash\underline{s}$ instead of $\vdash \underline{r}\backslash\underline{s}$ when the residuals are defined in an empty environment, and write $\#\underline{t}$ for the number of marks in $\underline{t}$.

If we have some term b, and we perform two distinct sets of redexes in it resulting in $b\backslash\underline{s} = c_1$ and $b\backslash\underline{t} = c_2$, we now need a way to conciliate these two terms and merge them back together, which can be done adapting of Lévy's cube lemma [Barendregt 1981] to the CPS-calculus:

Lemma 5.1 (Cube Property). *The residuals relation satisfies the cube property. Let $\underline{r}$, $\underline{s}$ and $\underline{t}$ be compatible marked terms; then $(\underline{r}\backslash\underline{s})\backslash(\underline{t}\backslash\underline{s}) = (\underline{r}\backslash\underline{t})\backslash(\underline{s}\backslash\underline{t})$.*

Proof. By induction and case analysis. □

This means that, to join c_1 and c_2, all we have to do is perform $\underline{t}\backslash\underline{s}$ in c_1 and $\underline{s}\backslash\underline{t}$ in c_2. We can now go back to the example given at the start of this section, taking $\underline{s} = \underline{k\langle\rangle}\ \{\ k\langle\rangle = j\langle\rangle\ \}\ \{\ j\langle\rangle = c\ \}$ and $\underline{t} = k\langle\rangle\ \{\ k\langle\rangle = \underline{j\langle\rangle}\ \}\ \{\ j\langle\rangle = c\ \}$. Noticing these two sets are compatible, we can verify:

- The residuals of $\underline{s}$ after $\underline{t}$ are defined as $\underline{s}\backslash\underline{t} = \underline{k\langle\rangle}\ \{\ k\langle\rangle = c\ \}\ \{\ j\langle\rangle = c\ \}$.
- The residuals of $\underline{t}$ after $\underline{s}$ are defined as $\underline{t}\backslash\underline{s} = \underline{j\langle\rangle}\ \{\ k\langle\rangle = \underline{j\langle\rangle}\ \}\ \{\ j\langle\rangle = c\ \}$.

The redexes marked above are precisely the ones needed to close the diagram.

5.2 Developments

Besides being able to find which jumps are necessary to conciliate two reduction paths, we also need to be sure that those jumps can actually be performed individually. For doing so, all we need to show is that, by partially performing any subset of the desired set, we will still get to the same resulting term.

Lemma 5.2 (Partial Development). *Subset of marks in a marked term may be partially reduced. Let* $\underline{r}$, $\underline{s}$ *and* $\underline{t}$ *be compatible marked term. If* $\underline{t} \subset \underline{s}$, *then* $\underline{r}\backslash\underline{s} = (\underline{r}\backslash\underline{t})\backslash(\underline{s}\backslash\underline{t})$.

In fact, not only will we reach the same result by picking any order of redexes, but we also always get there eventually: even though partially applying a set of redexes might copy some marks in a set, we can still check that, as no newly created redex can ever be marked, this cannot loop.

Theorem 5.3 (Finite Development). *Marks may be reduced in any order without looping. Let* $\mathcal{R}$ *be a notion of reduction on marked terms such that* $\underline{r}\ \mathcal{R}\ \underline{s}$ *is defined if and only if there is a* $\underline{t}$ *with* $\#\underline{t} > 0$, $\underline{t} \subset \underline{r}$, *and* $\underline{r}\backslash\underline{t} = \underline{s}$. *Then* $\mathcal{R}$ *is strongly normalizing.*

5.3 Parallel Reduction

The residuals relation is defined over sets of redexes. We turn it into a proper notion of reduction for terms, which we define as follows, by allowing any nonempty set of existing jumps to be simultaneously performed:

$\boxed{b \twoheadrightarrow b}$ *Parallel Reduction*

$$b \twoheadrightarrow c \quad = \quad \exists \underline{r},\ \#\underline{r} > 0 \ \wedge\ b\backslash\underline{r} = c$$

The remaining details are standard [Barendregt 1981; Yoshida et al. 2004]. We show that parallel reduction indeed is a notion of parallel reduction, which allows us to derive that $\twoheadrightarrow^* = \rightsquigarrow_j^*$, and hence confluence for $\rightsquigarrow_j$ from the cube property.

Lemma 5.4 (Parallel Moves). *Parallel reduction performs a sequence of one or more jumps.*

(1) If $b \rightsquigarrow_j c$, *then* $b \twoheadrightarrow c$.
(2) If $b \twoheadrightarrow c$, *then* $b \rightsquigarrow_j^+ c$.

Proof. Statement (1) follows by checking that $b\backslash\underline{r} = c$ with $\underline{r}$ being a mark for the single jump performed from b to c. Statement (2) follows from Lemma 5.2 and Theorem 5.3. □

In the above, the use of Lemma 5.2 and Theorem 5.3, is not actually necessary; it would be possible, for example, to just pick reduction from right to left instead (by induction on the number of marks).

Lemma 5.5 (Parallel Diamond). *The reflexive closure of parallel reduction satisfies the diamond property. I.e., if* $b \twoheadrightarrow^? c_1$ *and* $b \twoheadrightarrow^? c_2$, *then there's a* d *such that* $c_1 \twoheadrightarrow^? d$ *and* $c_2 \twoheadrightarrow^? d$.

Proof. Follows by case analysis and by Lemma 5.1. □

Theorem 5.6 (Confluence). *Full reduction is confluent. If* $b \rightsquigarrow^* c_1$ *and* $b \rightsquigarrow^* c_2$, *then there is a* d *such that* $c_1 \rightsquigarrow^* d$ *and* $c_2 \rightsquigarrow^* d$.

Proof. Follows by standard argument using Lemma 5.5 and Corollary 3.7. □

The usual corollary of confluence is that normal forms are unique. As such, we can guarantee that choosing any reduction strategy will still lead to the same result, as long as it finishes.

6 Factorization

Even though full reduction is confluent, this does not guarantee that every reduction strategy will lead to a normal form. Consider a term $\mathsf{k}\langle\rangle\ \{\ \mathsf{j}\langle\rangle = \Omega\ \}$, where Ω is given as $\mathsf{k}\langle\mathsf{k}\rangle\ \{\ \mathsf{k}\langle\mathsf{k}\rangle = \mathsf{k}\langle\mathsf{k}\rangle\ \}$, which is closed. This term has a normal form, $\mathsf{k}\langle\rangle$, that can be achieved by a garbage collection step. However, as $\Omega \rightsquigarrow \Omega$, it also contains an infinite reduction sequence.

Both λ-calculi have a notion of *normal order*, which is a reduction strategy that guarantees to lead to a normal form if one exists. Though the result for the standard λ-calculus is older [Barendregt 1981], Plotkin gives a proof for this by demonstrating that both calculi can reorder reductions into a *standard reduction sequence* by always performing redexes from left to right. From this it follows that the CBN and CBV reduction relations, though deterministic, are enough to reduce a term into a value, as the standard sequence always starts with them. In fact, this *standardization* result is used as a means to show a different but related result, *factorization* of head reduction, which says that any reduction sequence may be reordered to start with head steps [Accattoli et al. 2019]. So, as our final step for following Plotkin's program, we prove factorization of head reduction, meaning that any reduction sequence can be reordered such that it does every required head-redex first.

6.1 Postponing Inner Steps

We follow the method set out by Accattoli et al. [2019], which generalizes a technique given by Takahashi [1989] for proving factorization in the λ-calculus. As with confluence, we only need to reason about jump reductions because garbage collection is a shrinking reduction. The proof then follows directly from the theory of residuals which is used to sort head-redexes, as parallel reduction allows us to perform jumps in any order.

Two additional auxiliary notions of reduction are necessary: we write $\rightsquigarrow_{\mathsf{i}}$ for an *inner jump*, i.e., a single jump that occurs anywhere in a term other than head position, and we write $\twoheadrightarrow_{\mathsf{i}}$ for a notion of *parallel inner reduction*, where any jumps in a term that are not in head position can be simultaneously reduced. As we already have every necessary piece, as those were also used in the proof of confluence, we can proceed to prove factorization.

Theorem 6.1 (Factorization). *Reduction sequences can always be reordered such that they start with head reduction, followed by inner steps, and finally by garbage collection steps. So, if* $\mathsf{b} \rightsquigarrow^* \mathsf{c}$, *then* $\mathsf{b} \rightarrow^* \rightsquigarrow_{\mathsf{i}}^* \rightsquigarrow_{\mathsf{g}}^* \mathsf{c}$.

Proof. By Corollary 3.7, since $\mathsf{b} \rightsquigarrow^* \mathsf{c}$, then $\mathsf{b} \rightsquigarrow_{\mathsf{j}}^* \rightsquigarrow_{\mathsf{g}}^* \mathsf{c}$. Then the proof follows as given by Accattoli et al. [2019], by showing that $\{\rightarrow,\ \rightsquigarrow_{\mathsf{i}}\}$ is a *macro-step system*. Namely, it is enough to demonstrate that (1) $\rightsquigarrow_{\mathsf{j}} = \rightarrow \cup \rightsquigarrow_{\mathsf{i}}$, (2) $\rightsquigarrow_{\mathsf{i}} \subset \twoheadrightarrow_{\mathsf{i}}^? \subset \rightsquigarrow_{\mathsf{i}}^*$, (3) $\twoheadrightarrow_{\mathsf{i}}^? \rightarrow \subset \twoheadrightarrow^?$, and (4) $\twoheadrightarrow^? \subset \rightarrow^* \twoheadrightarrow_{\mathsf{i}}^?$. □

6.2 Plotkin's Program

We now have everything set to prove that the CPS-calculus is related to Appel's IR in the same sense that the λ-calculus is related to a functional programming language. We formally state this relationship as follows:

Corollary 6.2 (Foundation). *Appel's IR is related to the CPS-calculus in the sense of Plotkin [1975]. Namely, the following statements hold:*

(1) If $\mathsf{b} \Downarrow$ *then there are* c *and* k *such that* $\mathsf{b} = \mathsf{c}$ *and* $\mathsf{c} \downarrow_{\mathsf{k}}$.
(2) If $\mathsf{b} = \mathsf{c}$ *then, for any* C, $\mathsf{C}[\mathsf{b}] \Downarrow$ *if and only if* $\mathsf{C}[\mathsf{c}] \Downarrow$.

Proof. Both statements follow from Theorems 2.1 and 3.1 and the definition of bisimilarity. □

A careful reader might have noticed: everything we needed to prove they are related had already been given by Section 3. So why did we decide to postpone this result? The proof given by Merro

and Sangiorgi [2004] for Theorem 2.1 follows from a close correspondence between evaluation for a term b and its translation $b^\bullet$ into the asynchronous localized π-calculus. However, we also now have the necessary tools to prove this property internally in the CPS-calculus and to do so by following the approach set out by Plotkin.

We define $\rightleftharpoons$, which we call the *structural relation*, to be the least equivalence relation closed under contraction, floating, and an additional global rule (ETA'), admissible in the equational theory, given as follows:

$$\text{(ETA')} \qquad C[b \{ k\langle\vec{x}\rangle = j\langle\vec{x}\rangle \}] \quad = \quad C[b[j/k]] \qquad (\text{given } j \notin \vec{x},\ j \notin \mathtt{dom}(C), \text{ and } k \in \mathtt{FV}(b))$$

The above rule is not meant to be compatible in the structural relation (hence $\rightleftharpoons$ is not a congruence). Intuitively, we want to capture the subset of (ETA) that cannot be performed by garbage collection nor by contraction and jumping. With the aid of Lemma 7.5 (postponed till Section 7), the proof of Theorem 2.1 follows:

Proof for Theorem 2.1. Let $b_1 = b_2$ and $C[b_1] \to^* c \downarrow_k$. As equality is a congruence and by Lemma 3.3, $c = C[b_2]$. We proceed by first showing that, by Lemma 7.5, the relation $\mathcal{R} = \twoheadrightarrow^? \cup \rightsquigarrow_g^*$ is confluent modulo $\rightleftharpoons$, and, as a consequence, it is also Church-Rosser modulo $\rightleftharpoons$. As $\mathcal{R}^*/\rightleftharpoons$ coincides with $=$, there are terms d_1 and d_2 such that $c \rightsquigarrow^* d_1$, $C[b_2] \rightsquigarrow^* d_2$, and $d_1 \rightleftharpoons d_2$. As $\rightsquigarrow$ and $\rightleftharpoons$ are barb preserving, it follows that $d_2 \downarrow_k$. Then, by Theorem 6.1, $C[b_2] \to^* \rightsquigarrow_i^* \rightsquigarrow_g^* d_2 \downarrow_k$, and, as neither $\rightsquigarrow_i$ nor $\rightsquigarrow_g$ can change a jump in head position, $C[b_2] \to^* \downarrow_k$ as required. □

We have now a proof that the CPS-calculus is indeed a sound theoretical foundation for IRs such as Appel's and Kennedy's. As such, compiler developers are free to use the CPS-calculus as a tool for proving program transformations correct and otherwise reasoning about CPS terms.

7 Strong Normalization

Up to this point, our main focus was the *untyped* CPS-calculus. We recall that, at its inception, the CPS-calculus was studied in a categorical setting, and as such its type system was pervasively taken into account [Thielecke 1997a]. Our final goal is to investigate this type system for its properties and prove a similar result to that of many typed λ-calculi: reduction is strongly normalizing for well-typed terms. In other words, computation terminates regardless of evaluation strategy.

7.1 Simple Type System

The simple type system given by Thielecke [1997a] contains a single type constructor, namely a polyadic negation type, used to type continuations, with environments defined as usual. Base types, ranged over by X, are also considered, although they are not necessary.[11]

$$\textit{Types} \quad \tau \ ::= \ \neg\vec{\tau} \mid X \qquad\qquad \textit{Environments} \quad \Gamma \ ::= \ \cdot \mid \Gamma,\ x\colon\tau$$

We write, e.g., $\neg(\tau,\ \upsilon)$ for a binary instance of polyadic negation to make it clear the precedence of negation in the syntax. Type environments represent association lists between variables and types.

Typing judgements are represented by one-sided sequents. While environments assign types to each free variable in a term, there is no type associated to a term directly as they never return a value. Alternatively, terms may be seem as having a single fixed type, bottom, and a judgement asserts that its environment entails a contradiction. The typing rules are given as follows:

[11] Differently from the simply typed λ-calculus, base types are not needed in here as a base case since $\neg()$ is a valid type. However, base types require extra considerations which we deem interesting for the normalization proof.

$\boxed{\Gamma \vdash \mathsf{b}}$ *Simple Type System*

$$\frac{\Gamma(\mathsf{k}) = \neg\vec{\tau} \qquad \Gamma(\vec{\mathsf{x}}) = \vec{\tau}}{\Gamma \vdash \mathsf{k}\langle\vec{\mathsf{x}}\rangle}\ (\mathrm{J}) \qquad \frac{\Gamma,\ \mathsf{k}\colon \neg\vec{\tau} \vdash \mathsf{b} \qquad \Gamma,\ \vec{\mathsf{x}}\colon \vec{\tau} \vdash \mathsf{c}}{\Gamma \vdash \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{x}}\rangle = \mathsf{c}\ \}}\ (\mathrm{B})$$

Also, the usual structural rules, namely weakening, contraction and exchange, are admissible, as already stated by Thielecke [1997a]:

$$\frac{\Gamma \vdash \mathsf{b} \qquad \mathsf{x} \notin \mathsf{FV}(\mathsf{b})}{\Gamma,\ \mathsf{x}\colon \tau \vdash \mathsf{b}}\ (\mathrm{W}) \qquad \frac{\Gamma,\ \mathsf{x}\colon \tau,\ \mathsf{y}\colon \tau \vdash \mathsf{b}}{\Gamma,\ \mathsf{x}\colon \tau \vdash \mathsf{b}[\mathsf{x}/\mathsf{y}]}\ (\mathrm{C}) \qquad \frac{\Gamma,\ \mathsf{x}\colon \tau,\ \mathsf{y}\colon \upsilon,\ \Delta \vdash \mathsf{b}}{\Gamma,\ \mathsf{y}\colon \upsilon,\ \mathsf{x}\colon \tau,\ \Delta \vdash \mathsf{b}}\ (\mathrm{X})$$

As the CPS-calculus is polyadic and terms can get stuck, and as we have introduced a new notion of reduction, it is important to ensure that the type system is actually sound. This is proven in the standard way, through the progress and subject reduction properties.

Theorem 7.1 (Type Soundness). *The simple type system is sound. Assuming* $\Gamma \vdash \mathsf{b}$ *then we have:*

(1) Progress: either $\mathsf{b} \downarrow_{\mathsf{k}}$ *for some* k *or* $\mathsf{b} \longrightarrow \mathsf{c}$ *for some* c*. Thus,* b *is not stuck.*

(2) Subject reduction: if $\mathsf{b} \rightsquigarrow \mathsf{c}$ *for some* c*, then* $\Gamma \vdash \mathsf{c}$*.*

Note that statement (2) above works for full reduction, not for some arbitrary shrinking reduction.

7.2 Typed CPS Translation

While working in a typed setting, it may be desirable to also verify that a CPS translation is type preserving: if a term is typed in the source language, then it is also typed in the target. This is the case for the simply typed λ-calculus using both of Plotkin's translations presented in Section 4. We extend both translations to types and typing environments:

$$\textit{Functional Types}\quad A \ ::=\ A \to A \mid X$$

$$\begin{aligned}
[\![X]\!]_{\mathsf{N}} &= \neg X & \qquad [\![X]\!]_{\mathsf{V}} &= X \\
[\![A \to B]\!]_{\mathsf{N}} &= \neg\neg([\![A]\!]_{\mathsf{N}}, [\![B]\!]_{\mathsf{N}}) & \qquad [\![A \to B]\!]_{\mathsf{V}} &= \neg([\![A]\!]_{\mathsf{V}}, \neg[\![B]\!]_{\mathsf{V}}) \\
[\![\vec{\mathsf{x}}\colon \vec{A} \vdash \mathsf{e}\colon B]\!]_{\mathsf{N}} &= \vec{\mathsf{x}}\colon \overrightarrow{\neg[\![A]\!]_{\mathsf{N}}},\ \mathsf{k}\colon [\![B]\!]_{\mathsf{N}} \vdash [\![\mathsf{e}]\!]_{\mathsf{N}} & \qquad [\![\vec{\mathsf{x}}\colon \vec{A} \vdash \mathsf{e}\colon B]\!]_{\mathsf{V}} &= \vec{\mathsf{x}}\colon \overrightarrow{[\![A]\!]_{\mathsf{V}}},\ \mathsf{k}\colon \neg[\![B]\!]_{\mathsf{V}} \vdash [\![\mathsf{e}]\!]_{\mathsf{V}}
\end{aligned}$$

We omit the typing rules for the λ-calculus, as those are standard. Formally, we can now state type preservation for both translations as follow:

Theorem 7.2 (Type Preservation). *Both CPS translations are type preserving. If* $\vec{\mathsf{x}}\colon \vec{A} \vdash \mathsf{e}\colon B$*, then both* $[\![\vec{\mathsf{x}}\colon \vec{A} \vdash \mathsf{e}\colon B]\!]_{\mathsf{N}}$ *and* $[\![\vec{\mathsf{x}}\colon \vec{A} \vdash \mathsf{e}\colon B]\!]_{\mathsf{V}}$ *are derivable.*

Proof. By simple induction on the typing derivation and case analysis. □

Also, as we shall now proceed to prove that the simply typed CPS-calculus is strongly normalizing, we anticipate ourselves a bit by deriving a corollary, namely that we can derive that the simply typed λ-calculus is strongly normalizing as well.

Corollary 7.3. *The simply typed* λ*-calculus is strongly normalizing.*

Proof. Follows by Lemma 4.1, by using the CBN translation, and by Theorems 7.2 and 7.12. □

Although it's straightforward to derive that the simply typed λ_v-calculus is strongly normalizing too from the above result, since β_v-reduction is a strict subset of β-reduction, this may also be proven using the CBN translation and simulation results directly (instead of the CBV translation), by repeatedly applying a β_v-redex if one exists (note that, if $\mathsf{e} \to_{\beta_v} \mathsf{f}$, then $[\![\mathsf{e}]\!]_{\mathsf{N}} \rightsquigarrow^{+} [\![\mathsf{f}]\!]_{\mathsf{N}}$). In a sense, the CPS translations capture a notion of observational equivalence rather than a strict evaluation order, and a β_v-reduction step is sound for the CBN equivalence.

7.3 Reducibility at a Distance

The study of strong normalization is commonplace for functional calculi, usually motivated by the logical interpretation of terms. It has also been explored in the setting of process calculi: given that a client sends a message to a server, it is useful to be able to assert that the server will eventually provide an answer [Yoshida et al. 2004]. Similarly, it has been explored for explicit substitution calculi [Accattoli 2013, 2022]. But why would we care about strong normalization in an IR, though? As Kennedy [2005] puts it, if we know that a set of reductions is strongly normalizing, then we are allowed to apply them as we see fit during compilation, without worrying about an infinite loop.

The strong normalization result follows by adapting a well-known technique developed for the λ-calculus: Tait and Girard's reducibility method [Girard et al. 1989]. The basic idea of this technique is to define sets of untyped terms, called *reducibility predicates*, indexed by a type. The proof then holds by demonstrating that each predicate contains only strongly normalizing terms, and that each well-typed term is contained in the predicate for its type. However, the rigid syntax of the CPS-calculus imposes some technical difficulties.

The first issue that needs to be addressed is how to define each reducibility predicate, which are expected to be indexed over a type. Given a typing judgement, what should be considered the type of a term? Of course, we cannot consider only bottom as this would result in a single predicate, which would not help the proof go through. Instead we note that a reducibility predicate represents a logical relation that says that a term is well-behaved (in this case, strongly normalizing) when placed in any well-defined elimination context. In the λ-calculus, the elimination context for an abstraction is simply $[-]\ \mathsf{e}$, while in the CPS-calculus the elimination context for a jump $\mathsf{k}\langle\vec{\mathsf{x}}\rangle$ is any context C that binds the variable k to a continuation with the correct arity. Our main insight, then, is to work with reducibility at a distance: to represent well-defined elimination contexts for a jump, we have to simultaneously consider all the continuations that its context binds, and we can do so by indexing the reducibility candidate not by a type, but by an environment. Because of this, we call the reducibility predicate indexed over a term's environment its *semantic type*.

In order to perform a jump-redex $\mathsf{C}[\mathsf{k}\langle\vec{\mathsf{x}}\rangle]\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \} \rightsquigarrow_{\mathsf{j}} \mathsf{C}[\mathsf{c}[\vec{\mathsf{x}}/\vec{\mathsf{y}}]]\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}$ (as defined in Section 3.2), we can perform the exact same reasoning as in the proof for subject reduction in Theorem 7.1. We can apply several transformations on the environment for c: introducing variables with which c does not interact to account for the context C, reordering the bindings in the environment, and applying the substitutions for each parameter. These can be done, precisely, by the structural typing rules: weakening, exchange, and contraction. So, if we can show that semantic types are closed under structural rules, then we can prove strong normalization.

Although not a necessary component in the proof method by itself, proving that semantic types are closed under structural rules will require some auxiliary results in the case of the CPS-calculus, as its syntax is too constrained: in comparison to the λ-calculus, every value has to be named and there is no direct equivalent to a β-redex, and in comparison to the π-calculus, there is no scope extrusion rule or a terminal term 0 to reason about. Even so, it is possible to show that the required rewrites are admissible, by showing that strong normalization is closed under them. Just as in the adequacy result for CBV, though, it's possible that a larger notion of shrinking reduction could be used to simplify the following results.

7.4 Conservation

To show admissibility of structural rules on semantic types, we must first demonstrate some *conservation* results. Due to Corollary 3.7, we only need to focus on showing that jump reductions are strongly normalizing, as strong normalization for jump reduction and for full reduction coincide, and as such we simply refer to terms as strongly normalizing.

We draw attention to the fact that performing a jump cannot possibly erase an ill-behaved subterm, and as such, there are no *critical steps*: a step $b \rightsquigarrow_j c$ such that c is strongly normalizing while b is not. This property, which is called *conservation*, was first proved by Church for the λI-calculus [Barendregt 1981], where a β_I-redex has the form $(\lambda x.e)\ f$ with $x \in \mathsf{FV}(e)$, and shows that the existence of a normal form implies strong normalization. In the following, let $\mathtt{WN}$ be the set of terms in jump-normal form and $\mathtt{SN}$ be the set of strongly normalizing terms.

Theorem 7.4 (Conservation). *The following statements are both true and equivalent:*

(1) If $b \rightsquigarrow_j c$, *then* $b \in \mathtt{SN}$ *if and only if* $c \in \mathtt{SN}$.

(2) The sets $\mathtt{WN}$ *and* $\mathtt{SN}$ *coincide:* $\forall c,\ c \in \mathtt{WN}$ *if and only if* $c \in \mathtt{SN}$.

This property of jumps has been noticed before, and plays a key role in the strong normalization proof for the π-calculus by Yoshida et al. [2004]: the authors prove that a well-typed term has a normal form, and, from this, they derive that it is strongly normalizing. Though we remark that our method does not rely directly on this property, it is nevertheless of great help to prove the required conditions, and we use this property to yield simpler proofs for the following results.

Our goal of showing that semantic types admit structural rules will require a few additional conservation results. The structural relation given in Section 6 is also conservative in that it preserves the strong normalization property. As a proof technique, we introduce two auxiliary relations. We write $\twoheadrightarrow_d$ as the *full development* of a term: a deterministic relation such that $b \twoheadrightarrow_d c$ means that b reduces to c by performing all existing jumps in parallel, defined as $b \backslash \underline{r} = c$ with $\underline{r}$ being the nonempty maximal set of redexes in b (which is unique). We say that a relation $\mathcal{R}$ is a strong development simulation if $b_1 \twoheadrightarrow_d c_1$ and $b_1\ \mathcal{R}\ b_2$ imply that there is a c_2 such that $b_2 \twoheadrightarrow_d c_2$ and $c_1\ \mathcal{R}\ c_2$. If $\mathcal{R}$ and its converse relation are strong development simulations, which happens trivially if $\mathcal{R}$ is symmetric, then we say $\mathcal{R}$ is a strong development bisimulation. We then write $\dot{\sim}$ for *strong development bisimilarity*, defined as the largest strong development bisimulation, and prove that it includes the structural relation, and prove the corresponding conservation result.

Lemma 7.5 (Structural Bisimulation). *Strong development bisimulation is closed under the structural relation. So, if* $b \rightleftharpoons c$, *then* $b \dot{\sim} c$.

Proof. It is enough to show that $\rightleftharpoons$, which is symmetric, is a strong development simulation. □

Lemma 7.6 (Structural Conservation). *Structural rules preserve full normalization. So, if* $b \rightleftharpoons c$, *then* $b \in \mathtt{SN}$ *if and only if* $c \in \mathtt{SN}$.

Proof. It suffices to show one direction as $\rightleftharpoons$ is symmetric. By Lemma 5.4 and 7.4, strong normalization for $\rightsquigarrow_j$ and $\twoheadrightarrow_d$ coincide. As $\dot{\sim}$ is a strong bisimulation, it preserves reduction lengths thus if $b \dot{\sim} c$ and $b \in \mathtt{SN}$ then $c \in \mathtt{SN}$. By Lemma 7.5, from $b \rightleftharpoons c$ and $b \in \mathtt{SN}$, we have $c \in \mathtt{SN}$. □

A last additional conservation result is required. We need to show that if two terms are strongly normalizing under some elimination context, then we can compose them together within such a context, as long as they do not interact with each other (and, in a sense, they just run in parallel, similar to the π-calculus). In this case, we say that these two terms are *independent*.

Lemma 7.7 (Independence). *Strong normalization is preserved by composition of two independent terms in a static context. I.e., for any* C *static and any* D, *if* $C[b] \in \mathtt{SN}$, $C[D[c]] \in \mathtt{SN}$, $\mathsf{dom}(D) = \vec{x}$, *and* $k \notin \mathsf{FV}(b)$, *then* $C[b\ \{\ k\langle\vec{x}\rangle = c\ \}] \in \mathtt{SN}$.

7.5 Main Theorem

Our goal at last is to prove reducibility, which we can do by using the conservation results. We proceed to define semantic typing over typing environments, denoted by $[\![-]\!]$:

$$
\begin{aligned}
&(1) && [\![\cdot]\!] &&= \mathsf{SN} \\
&(2) && [\![\Gamma,\ \mathsf{k}\colon \neg\vec{\tau}]\!] &&= \{\ \mathsf{b} \mid \forall \mathsf{c} \in [\![\Gamma,\ \vec{\mathsf{x}}\colon \vec{\tau}]\!],\ \mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{x}}\rangle = \mathsf{c}\ \} \in [\![\Gamma]\!]\ \} \\
&(3) && [\![\Gamma,\ \mathsf{y}\colon X]\!] &&= \{\ \mathsf{b} \mid \forall \mathsf{x},\ \mathsf{k}\langle\mathsf{x}\rangle\ \{\ \mathsf{k}\langle\mathsf{y}\rangle = \mathsf{b}\ \} \in [\![\Gamma]\!]\ \}
\end{aligned}
$$

This definition is sound, as it follows by well-founded induction on the number of type terms in the environment. Each case progressively builds a well-defined elimination context around a term by inspecting the type of each variable in the environment. In the definition for (3), we note that base types cannot be eliminated, they may only be passed around to other continuations, and, as such, we just require that the term should still be valid by replacing y by any possible x by using a *delayed substitution* (as $\mathsf{k}\langle\mathsf{x}\rangle\ \{\ \mathsf{k}\langle\mathsf{y}\rangle = \mathsf{b}\ \} \rightsquigarrow^{+} \mathsf{b}[\mathsf{x}/\mathsf{y}]$).

We use an auxiliary lemma to simplify reasoning about which rules are admissible for semantic types. We say that a context C is *well-behaved* for an environment Γ if it binds a continuation for every variable with a negation type in Γ, and such continuation has the corresponding semantic type. As an example, say that C is well-behaved for some Γ. For any decomposition such that $\Gamma = \Delta_1, \mathsf{k}\colon \neg\vec{\tau},\ \Delta_2$, it follows that C binds the continuation k to some term c and $\mathsf{c} \in [\![\Delta_1]\!]$. By unfolding the definition of the semantic type and building a well-behaved context C for Γ, we can, to show that $\vec{\mathsf{b}} \in [\![\Gamma]\!]$ implies $\mathsf{c} \in [\![\Gamma]\!]$, simply show that $\mathsf{C}[\vec{\mathsf{b}}] \in \mathsf{SN}$ implies $\mathsf{C}[\mathsf{c}] \in \mathsf{SN}$.

Lemma 7.8 (Semantic Type Preservation). *Semantic types are closed under any transformation that preserves strong normalization under well-behaved contexts. Assume that, for every context C well-behaved for Γ, if $\mathsf{b} \in [\![\Gamma]\!]$ implies $\mathsf{C}[\mathsf{b}] \in \mathsf{SN}$ for any b, then $\mathsf{C}[\mathsf{c}] \in \mathsf{SN}$. It follows that $\mathsf{c} \in [\![\Gamma]\!]$.*

Admissibility of the structural rules on semantic types follows:

Lemma 7.9 (Structural Soundness). *Semantic types are preserved under structural rules. As such, the following statements hold:*

(1) Weakening: if $\mathsf{c} \in [\![\Gamma]\!]$ and $\mathsf{x} \notin \mathsf{FV}(\mathsf{c})$, then $\mathsf{c} \in [\![\Gamma,\ \mathsf{x}\colon \tau]\!]$.
(2) Exchange: if $\mathsf{c} \in [\![\Gamma,\ \mathsf{x}\colon \tau,\ \mathsf{y}\colon \upsilon,\ \Delta]\!]$, then $\mathsf{c} \in [\![\Gamma,\ \mathsf{y}\colon \upsilon,\ \mathsf{x}\colon \tau,\ \Delta]\!]$.
(3) Contraction: if $\mathsf{c} \in [\![\Gamma,\ \mathsf{x}\colon \tau,\ \mathsf{y}\colon \tau]\!]$, then $\mathsf{c}[\mathsf{x}/\mathsf{y}] \in [\![\Gamma,\ \mathsf{x}\colon \tau]\!]$.

Proof. The statements are proved by mutual induction with Lemma 7.10 on the number of type terms in the environment. For simplicity and without loss of generality, consider only negation types appearing in the environment, with the rules for base types proved in a similar way.

(1) Let $\mathsf{b} \in [\![\Gamma]\!]$ and $\mathsf{c} \in [\![\Gamma,\ \vec{\mathsf{y}}\colon \vec{\tau}]\!]$. Then, by Lemma 7.10 on a smaller environment, there is a context D such that $\mathsf{D}[\mathsf{c}] \in [\![\Gamma]\!]$ and $\mathsf{dom}(\mathsf{D}) = \vec{\mathsf{y}}$. By Lemma 7.8, we have to prove that $\mathsf{C}[\mathsf{b}] \in \mathsf{SN}$ and $\mathsf{C}[\mathsf{D}[\mathsf{c}]] \in \mathsf{SN}$ imply that $\mathsf{C}[\mathsf{b}\ \{\ \mathsf{k}\langle\vec{\mathsf{y}}\rangle = \mathsf{c}\ \}]$ for a fresh k, which follows by Lemma 7.7 by deriving a static context equivalent to C.

(2) Let $\mathsf{b} \in [\![\Gamma, \mathsf{x}\colon \neg\vec{\tau},\ \mathsf{y}\colon \neg\vec{\upsilon}]\!]$, $\mathsf{c} \in [\![\Gamma, \mathsf{x}\colon \neg\vec{\tau},\ \vec{\mathsf{z}}\colon \vec{\upsilon}]\!]$, and $\mathsf{d} \in [\![\Gamma,\ \mathsf{z}\colon \vec{\tau}]\!]$. Then, by statement (1), the inductive hypothesis on smaller environments, and Lemma 7.8, we have to show that $\mathsf{C}[\mathsf{b}\ \{\ \mathsf{y}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{x}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \{\ \mathsf{y}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}\ \}] \in \mathsf{SN}$ implies $\mathsf{C}[\mathsf{b}\ \{\ \mathsf{x}\langle\vec{\mathsf{z}}\rangle = \mathsf{d}\ \}\ \{\ \mathsf{y}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}] \in \mathsf{SN}$, which follows by Corollary 3.7 and Lemma 7.6.

(3) Let $\mathsf{b} \in [\![\Gamma,\ \mathsf{x}\colon \neg\vec{\tau},\ \mathsf{y}\colon \neg\vec{\tau}]\!]$ and $\mathsf{c} \in [\![\Gamma,\ \vec{\mathsf{z}}\colon \vec{\tau}]\!]$. Then, by statements (1) and (2) and by Lemma 7.8, we are left now to prove that $\mathsf{C}[\mathsf{b}\ \{\ \mathsf{y}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}\ \{\ \mathsf{x}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}] \in \mathsf{SN}$ implies that $\mathsf{C}[\mathsf{b}[\mathsf{y}/\mathsf{x}]\ \{\ \mathsf{y}\langle\vec{\mathsf{z}}\rangle = \mathsf{c}\ \}] \in \mathsf{SN}$, which follows by Corollary 3.7 and Lemma 7.6. □

We remark that the terms that appear in the cases for exchange and contraction correspond directly to instances of the (DISTR) and (CONTR) rules, respectively. For weakening, the proof follows by independence as two terms run in parallel without interacting, which acts like a refined version of (GC) where the unused continuation is well-behaved within its context. All that remains are the two pieces of the reducibility proof method: showing that each semantic type is reducible,

and thus is a nonempty set of strongly normalizing terms, and that every well-typed term is contained in its semantic type.

Lemma 7.10 (Reducibility). *Semantic types are reducible. For an arbitrary environment* Γ *then:*

(1) If $c \in [\![\Gamma]\!]$*, then* $c \in \mathsf{SN}$*.*
(2) There is a c *such that* $c \in [\![\Gamma]\!]$*.*

Proof. The statements are proved by mutual induction with Lemma 7.9 on the number of subtypes in the environment Γ, then by simple case analysis. □

Lemma 7.11 (Fundamental Lemma). *All well-typed terms are contained in their respective semantic types. So* $\Gamma \vdash c$ *implies that* $c \in [\![\Gamma]\!]$*.*

Proof. By induction on the typing judgement.
Case (B): $b\ \{\ k\langle\vec{y}\rangle = c\ \} \in [\![\Gamma]\!]$. Follows directly from the inductive hypotheses and from the definition of the semantic types.
Case (J): $k\langle\vec{x}\rangle \in [\![\Gamma]\!]$. By Lemma 7.8, there is a context C that is well-behaved for Γ, and knowing that for any $b \in [\![\Gamma]\!]$ it follows that $C[b] \in \mathsf{SN}$, we need to show that $C[k\langle\vec{x}\rangle] \in \mathsf{SN}$. As $\Gamma(k) = \neg\vec{\tau}$, there is a term c such that $\Gamma = \Delta_1,\ k\colon \neg\vec{\tau},\ \Delta_2$ and $c \in [\![\Delta_1,\ \vec{y}\colon \vec{\tau}]\!]$. By Lemma 7.9, we can use successive applications of exchange to treat semantic types up to permutation. We can use successive applications of weakening to show that $c \in [\![\Gamma,\ \vec{y}\colon \vec{\tau}]\!]$, and, from that, $c[\vec{x}/\vec{y}] \in [\![\Gamma]\!]$. By our hypothesis, it follows that $C[c[\vec{x}/\vec{y}]] \in \mathsf{SN}$, which is just one step away from our goal. Since C is well-formed for Γ and by Theorem 7.4, it follows that $C[k\langle x\rangle] \in \mathsf{SN}$ as required. □

We now state our last theorem, that full reduction is strongly normalizing in the simply typed CPS-calculus, meaning that there are no non-terminating reduction strategies in it.

Theorem 7.12 (Strong Normalization). *The simply typed CPS-calculus is strongly normalizing. For every well-typed term* $\Gamma \vdash c$*, it follows that* $c \in \mathsf{SN}$*.*

Proof. Follows directly from Lemmas 7.11 and 7.10. □

7.6 A Logic of Continuations

A usual consequence of strong normalization in a typed calculus is that logical consistency follows. This is also the case for the CPS-calculus, where consistency can be characterized simply by the absence of closed well-typed terms.

Corollary 7.13 (Consistency). *The simply typed CPS-calculus is consistent as a logic system. The following statements are both true and equivalent:*

(1) There is no well-typed term on an empty environment.
(2) There cannot be terms b *and* c *such that* $\vec{x}\colon \vec{\tau} \vdash b$ *and* $k\colon \neg\vec{\tau} \vdash c$*.*

Intuitively, as computation does not return, it instead stops when it does not know how to proceed. If a term is closed and it cannot be stuck, as is the case for well-typed terms, then it can always continue jumping indefinitely. Indeed, the fact that there cannot be any closed terms in the simple type system was already conjectured by Thielecke [1997a], and later proven by Levy [2001] in his PhD thesis by using a domain-theoretic approach, whereas we give a syntactical proof.

Consider, finally, the following example. Besides being capable of representing terms from the simply typed λ-calculus through the CPS translations given above in Section 4.1, the simply typed CPS-calculus is also capable of representing control effects. We can define `call/cc`, which is typed by Peirce's law for any types A and B [Griffin 1989], in CBV as follows:

$$\mathsf{k}\colon \neg[\![((A \to B) \to A) \to A]\!]_{\mathsf{V}} \vdash \mathsf{k}\langle \mathsf{f}\rangle \ \{\ \mathsf{f}\langle \mathsf{x},\ \mathsf{j}\rangle = \mathsf{x}\langle \mathsf{v},\ \mathsf{j}\rangle \ \{\ \mathsf{v}\langle \mathsf{y},\ \mathsf{h}\rangle = \mathsf{j}\langle \mathsf{y}\rangle\ \}\ \}$$

This demonstrates that the simply typed CPS-calculus is yet another constructive interpretation for classical propositional logic. However, it is an interpretation whose computational content has been used in compilers for the past three decades.

8 Conclusion

Throughout the history of programming language research, the λ-calculus has been a very important tool, established as both a prototypical functional language and as a foundational theory for functional programming. The π-calculus plays a similar role for concurrent programming languages, both as a prototype and a foundation. In this paper, we approached the question: could something similar be established for intermediate representations as used in production compilers? To answer that, we investigated the relationship between Thielecke's CPS-calculus, a theory of continuations motivated by categorical structure inherent in continuation-passing, and name-passing intermediate representations such as the one used in Appel's compiler, which are not closed under the usual notion of β-reduction.

By following the guidelines set by Plotkin, we give an affirmative answer to this question. In order to reach this result, we considered the abstract machine semantics given by Kennedy as the evaluator for his intermediate representation, and proposed a new reduction semantics for the CPS-calculus which is based on actual rewriting rules used in compilers as suggested by Appel, while separating these rules into computational steps, characterized by function inlining, and shrinking steps, meant to capture common optimizations such as dead-code elimination. Among our results, we proved that our reduction semantics is confluent and allows for the factorization of essential steps, which allows us to characterize the computational content of a term. As Kennedy's machine coincides with evaluation in the CPS-calculus, the notions of computational and behavioral equivalence in them are the same, a fact that we used to prove that the CPS-calculus is an adequate compilation target for both the λ-calculus and the λ_v-calculus by using both of Plotkin's CPS translations. This allows us to use the CPS-calculus as a tool for formally reasoning about optimizations, which we do by proving that contification preserves the meaning of a source program.

Finally, we studied the simply typed version of the CPS-calculus, as presented originally by Thielecke, showing that it is strongly normalizing for our notion of reduction. We did so by using a new proof method which is based on Tait and Girard's reducibility proof, but that reasons about reduction at a distance by using structural rules (weakening, exchange and contraction) on semantic types, which we hope can be adapted for proving strong normalization for other typed process calculi and explicit substitution calculi. As a standard consequence, the simply typed CPS-calculus is sound as a logic system, acting as a constructive interpretation for classical logic, meaning that the CPS-calculus not only may be used to prove correctness of optimizations within a verified compiler but may also work as a proof system itself. As future work, it's our goal to investigate more advanced type systems in the CPS-calculus, such as dependent and substructural types, using the metatheory developed here to study type-preserving compilation into an intermediate representation that was created from practical considerations.

Acknowledgments

We'd like to thank the anonymous reviewers for their helpful comments and suggestions. We also thank Beniamino Accattoli for an insightful discussion on proof techniques for the rewriting theory of calculi with reduction at a distance. The second author is supported in part by Schmidt Sciences.

References

Samson Abramsky. 1990. *The Lazy Lambda Calculus.* Addison-Wesley Longman Publishing Co., Inc., USA, 65–116.

Beniamino Accattoli. 2013. Linear Logic and Strong Normalization. In *24th International Conference on Rewriting Techniques and Applications (RTA 2013) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 21)*, Femke van Raamsdonk (Ed.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 39–54. https://doi.org/10.4230/LIPIcs.RTA.2013.39

Beniamino Accattoli. 2022. Exponentials as Substitutions and the Cost of Cut Elimination in Linear Logic. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science* (Haifa, Israel) *(LICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 49, 15 pages. https://doi.org/10.1145/3531130.3532445

Beniamino Accattoli, Eduardo Bonelli, Delia Kesner, and Carlos Lombardi. 2014. A Nonstandard Standardization Theorem. *SIGPLAN Not.* 49, 1 (Jan 2014), 659–670. https://doi.org/10.1145/2578855.2535886

Beniamino Accattoli, Claudia Faggian, and Giulio Guerrieri. 2019. Factorization and Normalization, Essentially. In *APLAS 2019 - 17th Asian Symposium on Programming Languages and Systems*. Springer, Bali, Indonesia. https://doi.org/10.1007/978-3-030-34175-6_9

Beniamino Accattoli and Delia Kesner. 2010. The Structural λ-Calculus. In *Computer Science Logic*, Anuj Dawar and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 381–395. https://doi.org/10.1007/978-3-642-15205-4_30

Amal Ahmed and Matthias Blume. 2011. An equivalence-preserving CPS translation via multi-language semantics. *SIGPLAN Not.* 46, 9 (sep 2011), 431–444. https://doi.org/10.1145/2034574.2034830

Roberto M. Amadio. 2017. *Operational methods in semantics (lecture notes).* https://api.semanticscholar.org/CorpusID:67205591 Accessed on June, 2024.

Roberto M. Amadio and Yann Régis-Gianas. 2012. Certifying and Reasoning on Cost Annotations of Functional Programs. In *Foundational and Practical Aspects of Resource Analysis*, Ricardo Peña, Marko van Eekelen, and Olha Shkaravska (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–89. https://doi.org/10.1007/978-3-642-32495-6_5

Andrew W. Appel. 1991. *Compiling with Continuations.* Cambridge University Press, USA. https://doi.org/10.1017/CBO9780511609619

Andrew W. Appel. 1998. SSA is Functional Programming. *SIGPLAN Not.* 33, 4 (apr 1998), 17–20. https://doi.org/10.1145/278283.278285

Andrew W. Appel and Trevor Jim. 1997. Shrinking Lambda Expressions in Linear Time. *J. Funct. Program.* 7, 5 (sep 1997), 515–540. https://doi.org/10.1017/S0956796897002839

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 233–246. https://doi.org/10.1145/199448.199507

Hendrik Pieter Barendregt. 1981. *The Lambda Calculus: Its Syntax and Semantics.* Elsevier Science Pub. Co. https://doi.org/10.2307/2274112

Olivier Savary Bélanger and Andrew W. Appel. 2017. Shrink fast correctly!. In *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming* (Namur, Belgium) *(PPDP '17)*. Association for Computing Machinery, New York, NY, USA, 49–60. https://doi.org/10.1145/3131851.3131859

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. 2004. Shrinking Reductions in SML.NET. In *Proceedings of the 16th International Conference on Implementation and Application of Functional Languages* (Lübeck, Germany) *(IFL'04)*. Springer-Verlag, Berlin, Heidelberg, 142–159. https://doi.org/10.1007/11431664_9

Nick Benton, Andrew Kennedy, and George Russell. 1998. Compiling standard ML to Java bytecodes. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming* (Baltimore, Maryland, USA) *(ICFP '98)*. Association for Computing Machinery, New York, NY, USA, 129–140. https://doi.org/10.1145/289423.289435

William J. Bowman, Youyou Cong, Nick Rioux, and Amal Ahmed. 2017. Type-Preserving CPS Translation of Σ and Π Types is Not Not Possible. *Proc. ACM Program. Lang.* 2, POPL, Article 22 (dec 2017), 33 pages. https://doi.org/10.1145/3158110

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or without? Whatever. *Proc. ACM Program. Lang.* 3, ICFP, Article 79 (Jul 2019), 28 pages. https://doi.org/10.1145/3341643

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Trans. Program. Lang. Syst.* 13, 4 (oct 1991), 451–490. https://doi.org/10.1145/115372.115320

Olivier Danvy, Belmina Dzafic, and Frank Pfennin. 1999. On proving syntactic properties of CPS programs. *Electronic Notes in Theoretical Computer Science* 26 (1999), 21–33. https://doi.org/10.1016/S1571-0661(05)80281-6 HOOTS '99, Higher Order Operational Techniques in Semantics.

Olivier Danvy and Julia Lawall. 1996. Back to Direct Style II: First-Class Continuations. *BRICS Report Series* RS-96-20 (1996).

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation* (Albuquerque,

New Mexico, USA) *(PLDI '93)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Carsten Führmann and Hayo Thielecke. 2004. On the call-by-value CPS transform and its semantics. *Information and Computation* 188, 2 (2004), 241–283. https://doi.org/10.1016/j.ic.2003.08.001

Jean-Yves Girard, Paul Taylor, and Yves Lafont. 1989. *Proofs and Types.* Cambridge University Press, USA. https://doi.org/10.2307/2274726

Timothy G. Griffin. 1989. A formulae-as-type notion of control. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 47–58. https://doi.org/10.1145/96709.96714

John Hatcliff and Olivier Danvy. 1997. Thunks and the λ-calculus. *Journal of Functional Programming - JFP* 7 (05 1997), 303–319. https://doi.org/10.1017/S0956796897002748

Kohei Honda and Olivier Laurent. 2010. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theoretical Computer Science* 411, 22 (2010), 2223–2238. https://doi.org/10.1016/j.tcs.2010.01.028

Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (San Francisco, California, USA) *(IR '95)*. Association for Computing Machinery, New York, NY, USA, 13–22. https://doi.org/10.1145/202529.202532

Andrew Kennedy. 2005. *[TYPES] normalization and programming languages.* http://lists.seas.upenn.edu/pipermail/types-list/2005/000970.html Sent to the [TYPES] mailing list. Accessed on July 11th, 2023.

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/1291151.1291179

Paul Blain Levy. 2001. *Call-by-push-value.* Ph. D. Dissertation. Queen Mary University of London, UK. https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.369233

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

Massimo Merro. 2010. On the Observational Theory of the CPS-Calculus. *Acta Inf.* 47, 2 (Mar 2010), 111–132. https://doi.org/10.1007/s00236-009-0112-9

Massimo Merro and Davide Sangiorgi. 2004. On Asynchrony in Name-Passing Calculi. *Mathematical. Structures in Comp. Sci.* 14, 5 (Oct 2004), 715–767. https://doi.org/10.1017/S0960129504004323

Robin Milner. 1992. Functions as processes. *Mathematical Structures in Computer Science* 2, 2 (1992), 119–141. https://doi.org/10.1017/S0960129500001407

Robin Milner and Davide Sangiorgi. 1992. Barbed Bisimulation. In *International Colloquium on Automata, Languages, and Programming*. Springer, 685–695. https://doi.org/10.1007/3-540-55719-9_114

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999. From system F to typed assembly language. *ACM Trans. Program. Lang. Syst.* 21, 3 (may 1999), 527–568. https://doi.org/10.1145/319301.319345

Marco Patrignani. 2020. Why should anyone use colours? or, syntax highlighting beyond code snippets. *arXiv preprint arXiv:2001.11334* (2020). https://doi.org/10.48550/arXiv.2001.11334

Benjamin C. Pierce and Davide Sangiorgi. 1993. Typing and subtyping for mobile processes. *Proceedings Eighth Annual IEEE Symposium on Logic in Computer Science* (1993), 376–385. https://doi.org/10.1017/S096012950007002X

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the λ-calculus. *Theor. Comput. Sci.* 1 (1975), 125–159. https://doi.org/10.1016/0304-3975(75)90017-1

John C. Reynolds. 1993. The Discoveries of Continuations. *LISP and Symbolic Computation* 6 (1993), 233–247. https://doi.org/10.1007/BF01019459

Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style. *SIGPLAN Lisp Pointers* V, 1 (Jan 1992), 288–298. https://doi.org/10.1145/141478.141563

Amr Sabry and Philip Wadler. 1997. A Reflection on Call-by-Value. *ACM Trans. Program. Lang. Syst.* 19, 6 (nov 1997), 916–941. https://doi.org/10.1145/267959.269968

Davide Sangiorgi and David Walker. 2001. *Pi-Calculus: A Theory of Mobile Processes.* Cambridge University Press, USA. https://doi.org/10.2178/bsl/1182353926

Guy L. Steele. 1978. *Rabbit: A Compiler for Scheme.* Technical Report. USA.

Masako Takahashi. 1989. Parallel reductions in λ-calculus. *Journal of Symbolic Computation* 7, 2 (1989), 113–123. https://doi.org/10.1016/S0747-7171(89)80045-8

Hayo Thielecke. 1997a. *Categorical Structure of Continuation Passing Style.* Ph. D. Dissertation. University of Edinburgh.

Hayo Thielecke. 1997b. Continuation Semantics and Self-adjointness. *Electronic Notes in Theoretical Computer Science* 6 (1997), 348–364. https://doi.org/10.1016/S1571-0661(05)80149-5

Paulo Torrens. 2024. *Artifact for "On the Operational Theory of the CPS-calculus".* https://doi.org/10.5281/zenodo.11498450

Philip Wadler. 2003. Call-by-value is dual to call-by-name. *SIGPLAN Not.* 38, 9 (aug 2003), 189–201. https://doi.org/10.1145/944746.944723

Nobuko Yoshida, Martin Berger, and Kohei Honda. 2004. Strong normalisation in the π-calculus. *Information and Computation* 191, 2 (2004), 145–202. https://doi.org/10.1016/j.ic.2003.08.004