BRICS

Basic Research in Computer Science

BRICS RS-00-46 Z. Yang: Reasoning About Code-Generation in Two-Level Languages

Reasoning About Code-Generation in Two-Level Languages

Zhe Yang

BRICS Report Series **RS-00-46**

ISSN 0909-0878 **December 2000**

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/46/`

Reasoning about Code Generation in Two-Level Languages

Zhe Yang*†
Department of Computer Science, New York University
251 Mercer Street, New York, NY 10012, USA
E-mail: zheyang@cs.nyu.edu

Abstract

We show that two-level languages are not only a good tool for describing code-generation algorithms, but a good tool for reasoning about them as well. Indeed, some general properties of two-level languages capture common proof obligations of code-generation algorithms in the form of two-level programs.

- To prove that the generated code behaves as desired, we use an *erasure* property, which equationally relates the generated code to an erasure of the original two-level program in the object language, thereby reducing the two-level proof obligation to a simpler one-level obligation.
- To prove that the generated code satisfies certain syntactic constraints, e.g., that it is in some normal form, we use a *type-preservation* property for a refined type system that enforces these constraints.

In addition, to justify concrete implementations of code-generation algorithms in one-level languages, we use a *native embedding* of a two-level language into a one-level language.

We present two-level languages with these properties both for a call-by-name object language and for a call-by-value object language with computational effects. Indeed, it is these properties that guide our language design in the call-by-value case. We consider two classes of non-trivial applications: one-pass transformations into continuation-passing style and type-directed partial evaluation for call-by-name and for call-by-value.

Keywords. Two-level languages, erasure, type preservation, native implementation, partial evaluation.

*This work was carried out at BRICS (Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation).

†Contact address: BRICS, Dept. of Computer Science, University of Aarhus, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. Phone: +45 8942 3109. Fax: +45 8942 3255. Email: zheyang@brics.dk

Contents

1 Introduction **4**
1.1 Background . . . 4
1.2 This work . . . 6

2 The call-by-name two-level language nPCF^2 **7**
2.1 Syntax and semantics . . . 7
2.2 Example: the CPS transformation . . . 10
2.3 Semantic correctness of the generated code: erasure . . . 10
2.4 Embedding nPCF^2 into a one-level language with a term type . . . 12
2.5 Example: call-by-name type-directed partial evaluation . . . 15
2.6 Syntactic correctness of the generated code: type preservation . . . 17

3 The general framework **19**

4 The call-by-value two-level language vPCF^2 **20**
4.1 Design considerations . . . 20
4.2 Syntax, semantics, and properties . . . 21
4.3 Example: call-by-value type-directed partial evaluation . . . 24

5 Related work **26**
5.1 Two-level formalisms for compiler construction . . . 27
5.2 Correctness of partial evaluators . . . 28
5.3 Macros and syntactic abstractions . . . 29
5.4 Multi-level languages . . . 30
5.5 Applications . . . 31

6 Conclusions **31**
6.1 Summary of contributions . . . 31
6.2 Direction for future work . . . 33

A Call-by-name CPS translation **34**

B Expanded proofs for nPCF^2 **35**
B.1 Type preservation and annotation erasure . . . 35
B.2 Native embedding . . . 35
B.3 Call-by-name type-directed partial evaluation . . . 43

C Call-by-value two-level language vPCF^2**: detailed development** **45**
C.1 Type preservation . . . 46
C.2 Determinacy . . . 50
C.3 Annotation erasure . . . 54
C.4 Native implementation . . . 55
C.4.1 A more "realistic" language: $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ . . . 55
C.4.2 The implementation language: $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$ . . . 57
C.4.3 Native embedding . . . 58

C.5 Call-by-value type-directed partial evaluation 66
C.5.1 Semantic correctness 66
C.5.2 Syntactic correctness 67

D Notation and symbols **69**

List of Figures

1 Base syntactic constituents 7
2 The two-level call-by-name language nPCF^2 8
3 The one-level call-by-name language nPCF 9
4 Call-by-value CPS transformation 11
5 Call-by-name type-directed partial evaluation 16
6 Inference rules for terms in long $\beta\eta$-normal form 18
7 nPCF^2-terms that generate code in long $\beta\eta$-normal form 18
8 The type system of vPCF^2 22
9 The evaluation semantics of vPCF^2 23
10 Call-by-value type-directed partial evaluation 25
11 vPCF^2-terms that generate code in λ_c-normal form 27
12 Inference rules for terms in λ_c-normal form 27
13 Call-by-name CPS transformation 34
14 One-level call-by-value language vPCF 53
15 Changes of $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ over vPCF^2 56
16 ML implementation of $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$-primitives 58
17 The evaluation semantics of $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$ 59

1 Introduction

1.1 Background

Programs that generate code, such as compilers and program transformers, appear everywhere, but it is often a demanding task to write them, and an even more demanding task to reason about them. The programmer needs to maintain a clear distinction between two languages of different binding times: the *static* (compile-time, meta) one in which the code-generation program is written, and the *dynamic* (run-time, object) one in which the generated code is written. To reason about code-generation programs, one always considers, at least informally, invariants about the code generated, e.g., that it type checks.

Two-level languages provide intuitive notations for writing code-generation programs succinctly. They incorporate both static constructs and dynamic constructs for modeling the binding-time separation. Their design usually considers certain semantic aspects of the object languages. For example, the typing safety of a two-level language states not only that (static) evaluation of well-typed programs does not go wrong, but also that the generated code is well-typed in the object language. The semantic benefit, however, often comes at the price of implementation efficiency and its related correctness proof.

Semantics vs. implementation: Consider, for example, the pure simply typed λ-calculus as the object language. A possible corresponding two-level language could have the following syntax.

$$E ::= x \mid \lambda x.E \mid E_1\, E_2 \mid \underline{\lambda} x.E \mid E_1 \underline{@} E_2$$

Apart from the standard (static) constructs, there are two dynamic constructs for building object-level expressions: $\underline{\lambda} x.E$ for λ-abstractions, and $E_1 \underline{@} E_2$ for applications. As a first approximation, one can think of the type of object expressions as an algebraic data type

```
E = VAR of string | LAM of string * E | APP of E * E
```

where $\underline{\lambda} x.E$ is shorthand for `LAM("`x`", `E`)`, $E_1 \underline{@} E_2$ is shorthand for `APP(`E_1`,`E_2`)`, and an occurrence of $\underline{\lambda}$-bound variable x is shorthand for `VAR("`x`")`. For instance, the term $\underline{\lambda} x.x$ is represented by `LAM("`x`", VAR "`x`")`.

This representation, by itself, does not treat variable binding in the object language. For instance, we can write a code transformer that performs η-expansion as $eta \triangleq \lambda f.\underline{\lambda} x.f \underline{@} x$, in the two-level language. Applying this code transformer to object terms with free occurrences of x exposes the problem that evaluation could capture names: For instance, evaluating $\underline{\lambda} x.eta\, x$ yields the object term $\underline{\lambda} x.\underline{\lambda} x.x \underline{@} x$, which is wrong and not even typable in the simply typed lambda calculus.

If we are working in a standard, high-level operational semantics that describes evaluation as symbolic computations on the two-level terms, then the solution to the name-capturing problem is simple: Dynamic λ-bound variables, like usual bound variables, should be subject to renaming during a non-capturing substitution $E\{E'/x\}$ (which is used in the evaluation of static applications). Therefore, in the earlier example, the two-level term $\underline{\lambda} x.(\lambda f.\underline{\lambda} x.f \underline{@} x)x$ does not evaluate to $\underline{\lambda} x.\underline{\lambda} x.x \underline{@} x$, but to $\underline{\lambda} x.\underline{\lambda} y.x \underline{@} y$. This precise issue is referred to as "hygienic macro expansion" in Kohlbecker's work [27, 28].

Indeed, the analogy between the dynamic λ-bound variables and the static λ-bound variables has long been adopted in the traditional, *staging view* of two-level languages, which is shaped by the pioneering work of Jones et al. [23, 24, 25]: Serving as an intermediate language of offline partial evaluators, a two-level language is the staged version of a corresponding one-level language. In this context, in addition to typing safety, another property, which we call *annotation erasure*, is important for showing the correctness of partial evaluators: The result of two-level evaluation has the same semantics as the unstaged version of the program. Taking up the earlier example again, we can see that the unstaged version of $\underline{\lambda} x.(\lambda f.\underline{\lambda} x.f \underline{@} x)x$, i.e., $\lambda x.(\lambda f.\lambda x.f\,x)x$, is β-equivalent to the generated term $\lambda x.\lambda y.x\,y$. In the symbolic framework, it is relatively easy to establish annotation erasure, at least in a call-by-name, effect-free setting.

For realistic implementations of two-level languages, capture-avoiding substitution is expensive and undesirable. Indeed, most implementations use some strategy to generate variables such that they do not conflict with each other. Unsurprisingly, it is more difficult to reason about these implementations. In fact, existing work that proved annotation erasure while taking the name generation into account used denotational-semantics formulations and stayed clear of operational semantics [12, 16, 36] (see Section 5.2 for detail).

Hand-written two-level programs: In the 1990s, two-level languages started to be used in expressing code-generation algorithms independently of dedicated partial evaluators. Such studies propel a second view of two-level languages: They are simply *one-level languages equipped with a code type that represents object-level terms*. This code-type view of two-level languages leads to two separate tracks of formal studies, again reflecting the tension between semantics and implementation.

The first track explores the design space of more expressive such languages, while retaining typing safety. Davies and Pfenning characterized multi-level languages in terms of temporal logic [10] and of modal logic [11]. Their work fostered the further development of multi-level languages such as MetaML [37]. In general, this line of work employs high-level operational semantics, in particular capture-free substitution, to facilitate a more conceptual analysis of design choices.

The second track uses the staging intuitions of two-level languages as a guide for finding new, higher-order code-generation algorithms; for the sake of efficiency, the algorithms are then implemented in existing (one-level) functional languages, using algebraic data types to encode the code types and generating names explicitly. As an example, Danvy and Filinski have used an informal two-level language to specify a one-pass CPS transformation that generates no administrative redexes [7], which is an optimization of Plotkin's original CPS transformation [45]. Similarly, a study of binding-time coercions by two-level eta-expansion has led Danvy to discover type-directed partial evaluation (TDPE), an efficient way to embed a partial evaluator into a pre-existing evaluator [5]. The proofs of correctness in both applications, as in the case of annotation erasure, stayed clear of two-level languages.

The case of TDPE deserves some interest of its own: Filinski formalized TDPE as a normalization process for terms in an *unconventional* two-level language, where the binding-time separation does not apply to all program constructs, but to only constants.[1] Using denotational semantics, he characterized the native implementability of TDPE in a conventional functional language [12, 13]. On the other hand, the intuitive connection between TDPE and conventional two-level languages has not been formalized.

1.2 This work

Our thesis is that *(1) we can formally connect the high-level operational semantics and the efficient, substitution-free implementation, and by doing so (2) we can both reason about code-generation algorithms directly in two-level languages and have their efficient and provably correct implementations.*

First, to support high-level reasoning, we equip the two-level language, say L^2, with a high-level operational semantics, which, in particular, embodies capture-avoiding substitution that takes dynamic λ-bound variables into account. We use the semantics to give simple, syntactic proofs of general properties such as annotation erasure, which reflects the staging view, and type preservation, which reflects the code-type view. In turn, we use these properties to prove semantic correctness of the generated code (i.e., it satisfies certain extensional properties) and syntactic correctness of the generated code (i.e., it satisfies certain intensional, syntactic constraints).

Next, to implement L^2-programs efficiently in a conventional one-level language (e.g., ML), we show a native embedding of L^2 into the implementation language. This native embedding provides efficient substitution-free implementation for the high-level semantics.

Overview of the paper The remainder of this paper fleshes out the preceding ideas with two instances of the framework. The first is a canonical two-level language nPCF^2 for a call-by-name object language (call-by-**n**ame "PCF of two-level languages", following Moggi [36]). The second, designed from scratch while taking the aforementioned properties (in particular, annotation erasure and native implementability) into account, is a more practically relevant two-level language vPCF^2: one with an instance of Moggi's call-by-**v**alue computational λ-calculus as its object language.

In Section 2 we present nPCF^2 together with its related one-level language nPCF, prove its properties, and apply them to the example of CPS transformations and call-by-name TDPE. From this study we abstract out, in Section 3,

[1] Without constants, the call-by-name version of TDPE coincides with Berger and Schwichtenberg's notion of normalization by evaluation [2].

our general framework, in particular the desired properties and the corresponding proof obligations they support. With this framework in mind, in Section 4 we design vPCF^2, prove its properties, and apply them to the example of call-by-value TDPE. We present the related work in Section 5 and conclude this partn Section 6. The detailed proofs and development are given in the appendices.

Notational conventions: Because we consider several different languages, we write $\mathsf{L} \vdash J$ to assert a judgment J in the language L, or we write simply J when L is clear from the context. We write $\equiv$ for strict syntactic equality, and $\sim_\alpha$ for equality up to α-conversion. Operations (syntactic translations) defined on types τ, say $\{\!|\tau|\!\}$, are homomorphically extended to apply to contexts: $\{\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!\} \equiv x_1 : \{\!|\tau_1|\!\}, \ldots, x_n : \{\!|\tau_n|\!\}$. A type-preserving translation $\{\!|-|\!\}$ of terms-in-contexts in language L_1 into ones in language L_2 is declared in the form $\boxed{\mathsf{L}_1 \vdash \Delta \rhd E : \sigma} \Longrightarrow \boxed{\mathsf{L}_2 \vdash \{\!|\Delta|\!\} \rhd \{\!|E|\!\} : \{\!|\sigma|\!\}}$. Meta-variables τ, σ, Γ, and Δ respectively range over two-level types, one-level types, two-level contexts, and one-level contexts.

2 The call-by-name two-level language nPCF^2

We present a canonical call-by-name (CBN) two-level language nPCF^2 (Section 2.1), cast the example of a one-pass CPS transformation as an nPCF^2 program (Section 2.2), and use an erasure argument to prove its correctness (Section 2.3). Building on a native embedding of nPCF^2 into a conventional language (Section 2.4), we formulate CBN TDPE in nPCF^2 and show its semantic correctness as well as its syntactic correctness (Sections 2.5 and 2.6).

2.1 Syntax and semantics

Base types $\mathsf{b} \in \mathbb{B}$: bool (boolean type), int (integer type)
Literals ℓ: $\mathbb{L}(\mathsf{bool}) = \{\mathsf{tt}, \mathsf{ff}\}, \mathbb{L}(\mathsf{int}) = \{\ldots, -1, 0, 1, \ldots\}$
Binary operators $\otimes$: $+, - : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$,
$=, < : \mathsf{int} \times \mathsf{int} \to \mathsf{bool}$

Figure 1: Base syntactic constituents

For the various languages in this article, we fix a set of base syntactic constituents (Figure 1). Figure 2 shows the type system ($\Gamma \rhd E : \tau$) and the evaluation semantics ($E \Downarrow V$) of nPCF^2 over a signature of typed constants $d : \sigma$ in the object language. For example, for the conditional construct, we can have a family of object-level constants $\mathsf{if}_\sigma : \mathsf{bool} \to \sigma \to \sigma$ in Sg.

In addition to the conventional CBN static part,[2] the language nPCF^2 has

[2]We omit product types but it is straightforward to add them and will not affect the results below.

a. **The object-level signature** Sg is a set of (uninterpreted) typed constants $d : \sigma$ in the object language.

b. **Syntax**

Types $\tau ::= \mathsf{b} \mid \bigcirc\sigma \mid \tau_1 \to \tau_2$ (two-level types)

$\sigma ::= \mathsf{b} \mid \sigma_1 \to \sigma_2$ (object-code types)

Contexts $\Gamma ::= \cdot \mid \Gamma, x : \tau$

Raw terms $E ::= \ell \mid x \mid \lambda x.E \mid E_1\, E_2 \mid \mathbf{fix}\, E \mid \mathbf{if}\, E_1\, E_2\, E_3$
$\mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid \underline{d} \mid \underline{\lambda} x.E \mid E_1 \underline{@} E_2$

Typing Judgment $\boxed{\mathsf{nPCF}^2 \vdash \Gamma \rhd E : \tau}$

(Static)

$$[\mathit{lit}]\ \frac{\ell \in \mathbb{L}(\mathsf{b})}{\Gamma \rhd \ell : \mathsf{b}} \qquad [\mathit{var}]\ \frac{x : \tau \in \Gamma}{\Gamma \rhd x : \tau}$$

$$[\mathit{lam}]\ \frac{\Gamma, x : \tau_1 \rhd E : \tau_2}{\Gamma \rhd \lambda x.E : \tau_1 \to \tau_2} \qquad [\mathit{app}]\ \frac{\Gamma \rhd E_1 : \tau_2 \to \tau \quad \Gamma \rhd E_2 : \tau_2}{\Gamma \rhd E_1\, E_2 : \tau}$$

$$[\mathit{fix}]\ \frac{\Gamma \rhd E : \tau \to \tau}{\Gamma \rhd \mathbf{fix}\, E : \tau} \qquad [\mathit{if}]\ \frac{\Gamma \rhd E_1 : \mathsf{bool} \quad \Gamma \rhd E_2 : \tau \quad \Gamma \rhd E_3 : \tau}{\Gamma \rhd \mathbf{if}\, E_1\, E_2\, E_3 : \tau}$$

$$[\mathit{bop}]\ \frac{\Gamma \rhd E_1 : \mathsf{b}_1 \quad \Gamma \rhd E_2 : \mathsf{b}_2}{\Gamma \rhd E_1 \otimes E_2 : \mathsf{b}} (\otimes : \mathsf{b}_1 \times \mathsf{b}_2 \to \mathsf{b})$$

(Dynamic)

$$[\underline{\mathit{lift}}]\ \frac{\Gamma \rhd E : \mathsf{b}}{\Gamma \rhd \$_{\mathsf{b}} E : \bigcirc\mathsf{b}} \qquad [\underline{\mathit{cst}}]\ \frac{Sg(d) = \sigma}{\Gamma \rhd \underline{d} : \bigcirc\sigma}$$

$$[\underline{\mathit{lam}}]\ \frac{\Gamma, x : \bigcirc\sigma_1 \rhd E : \bigcirc\sigma_2}{\Gamma \rhd \underline{\lambda} x.E : \bigcirc(\sigma_1 \to \sigma_2)} \qquad [\underline{\mathit{app}}]\ \frac{\Gamma \rhd E_1 : \bigcirc(\sigma_2 \to \sigma) \quad \Gamma \rhd E_2 : \bigcirc\sigma_2}{\Gamma \rhd E_1 \underline{@} E_2 : \bigcirc\sigma}$$

c. **Evaluation Semantics** $\boxed{\mathsf{nPCF}^2 \vdash E \Downarrow V}$

Values $V ::= \ell \mid \lambda x.E \mid \mathcal{O}$

$\mathcal{O} ::= \$_{\mathsf{b}}\ell \mid x \mid \underline{\lambda} x.\mathcal{O} \mid \mathcal{O}_1 \underline{@} \mathcal{O}_2 \mid \underline{d}$

(Static)

$$[\mathit{lit}]\ \frac{}{\ell \Downarrow \ell} \qquad [\mathit{lam}]\ \frac{}{\lambda x.E \Downarrow \lambda x.E} \qquad [\mathit{app}]\ \frac{E_1 \Downarrow \lambda x.E' \quad E'\{E_2/x\} \Downarrow V}{E_1\, E_2 \Downarrow V}$$

$$[\mathit{fix}]\ \frac{E(\mathbf{fix}\, E) \Downarrow V}{\mathbf{fix}\, E \Downarrow V} \qquad [\mathit{if}\text{-}\mathsf{tt}]\ \frac{E_1 \Downarrow \mathsf{tt} \quad E_2 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V} \qquad [\mathit{if}\text{-}\mathsf{ff}]\ \frac{E_1 \Downarrow \mathsf{ff} \quad E_3 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V}$$

$$[\otimes]\ \frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{E_1 \otimes E_2 \Downarrow V} (V_1 \otimes V_2 = V)$$

(Dynamic)

$$[\underline{\mathit{lift}}]\ \frac{E \Downarrow \ell}{\$_{\mathsf{b}} E \Downarrow \$_{\mathsf{b}}\ell} \qquad [\underline{\mathit{var}}]\ \frac{}{x \Downarrow x} \qquad [\underline{\mathit{cst}}]\ \frac{}{\underline{d} \Downarrow \underline{d}}$$

$$[\underline{\mathit{lam}}]\ \frac{E \Downarrow \mathcal{O}}{\underline{\lambda} x.E \Downarrow \underline{\lambda} x.\mathcal{O}} \qquad [\underline{\mathit{app}}]\ \frac{E_1 \Downarrow \mathcal{O}_1 \quad E_2 \Downarrow \mathcal{O}_2}{E_1 \underline{@} E_2 \Downarrow \mathcal{O}_1 \underline{@} \mathcal{O}_2}$$

Figure 2: The two-level call-by-name language nPCF^2

Types	σ	$::=$	$\mathsf{b} \mid \sigma_1 \rightarrow \sigma_2$
Raw terms	E	$::=$	$\ell \mid x \mid \lambda x.E \mid E_1\, E_2 \mid d$
			$\mid \mathbf{fix}\, E \mid \mathbf{if}\, E_1\, E_2\, E_3 \mid E_1 \otimes E_2$
Contexts	Δ	$::=$	$\cdot \mid \Delta, x : \sigma$

Typing Judgment $\boxed{\mathsf{nPCF} \vdash \Delta \rhd E : \sigma}$

The static part of nPCF^2 plus: $[cst]\,\dfrac{Sg(d) = \sigma}{\Delta \rhd d : \sigma}$

Equational Rules $\boxed{\mathsf{nPCF} \vdash \Delta \rhd E_1 = E_2 : \sigma}$

The congruence rules, $[\beta]$, $[\eta]$, and equations for **fix**, **if**, and binary operators $\otimes$. (omitted)

Figure 3: The one-level call-by-name language nPCF

a family of code types $\bigcirc\sigma$, indexed by the types σ of the represented object terms, and their associated constructors, which we call the *dynamic constructs*. For example, in the base case, dynamic constants $\underline{d} : \bigcirc\sigma$ represent the corresponding constants $d : \sigma$ in the object language; static values of base types b, called the literals, can be "lifted" into the code types $\bigcirc\mathsf{b}$ with $\$_{\mathsf{b}}$, so that the result of static evaluation can appear in the generated code. The dynamic constructs are akin to data constructors of the familiar algebraic types, but with the notable exception that the dynamic λ-abstraction is a binding operator: As mentioned in the introduction, the variables introduced are, like usual bound variables, subject to renaming during a non-capturing substitution $E\{E'/x\}$ (which is used in the evaluation of static applications).

The evaluation judgment of the form $E \Downarrow V$ reads that evaluation of the term E leads to a *value* V. Evaluation is deterministic modulo α-conversion: If $E \Downarrow V_1$ and $E \Downarrow V_2$ then $V_1 \sim_\alpha V_2$. A value can be a usual static value (literal or λ-abstraction) or a code-typed value $\mathcal{O}$. Code-typed values are in 1-1 correspondence with raw λ-terms in the object language by erasing their annotations (erasure will be made precise in Section 2.3).

Because evaluation proceeds under dynamic λ-abstractions, intermediate results produced during the evaluation can contain free dynamic variables [37]. Properties about the evaluation, therefore, are usually stated on terms which are closed on static variables, but not necessarily dynamic variables. For example, a standard property of evaluation in two-level languages is type preservation for *statically closed terms*.

Theorem 2.1 (Type preservation). *If $\bigcirc\Delta \rhd E : \tau$ and $E \Downarrow V$, then $\bigcirc\Delta \rhd V : \tau$. ($\bigcirc\Delta$ is the element-wise application of the $\bigcirc(-)$ constructor to the context Δ.)*

The proof, as most proofs for this part, can be found in the appendices.

As a consequence of this theorem, if $\bigcirc\Delta \rhd E : \bigcirc\sigma$ holds, then $E \Downarrow V$ implies that V is of the form $\mathcal{O}$.

Figure 3 shows the corresponding one-level language nPCF. The language includes not only the constructs of the object language, but also the static constructs of nPCF^2. Though the static constructs will not appear in the generated code, they are needed to specify and prove the semantic correctness of the generated code.

The equational theory of nPCF is standard for CBN languages. We only note that there are no equational rules for the constants d in the object language, thereby leaving them uninterpreted. That is, any interpretation of these constants is a a model of nPCF.

2.2 Example: the CPS transformation

Our first example is the typed versions of two transformations of the pure, simply typed, call-by-value λ-calculus (Figure 4a) into continuation-passing style (CPS).[3] The typed formulation [32] of Plotkin's original transformation [45] maps a term E directly into a one-level term $\{\!|E|\!\}_{\mathsf{p}\kappa}$ (Figure 4b), but it generates a lot of administrative redexes—roughly all the bindings named k introduce an extra redex—and to remove these redexes requires a separate pass. Danvy and Filinski's one-pass CPS transformation instead maps the term into a two-level program $\{\!|E|\!\}_{\mathsf{df}\kappa}$ (Figure 4c); evaluating $\{\!|E|\!\}_{\mathsf{df}\kappa}$ produces the resulting CPS term. The potential administrative redexes are annotated as static, and thus are reduced during the evaluation of $\{\!|E|\!\}_{\mathsf{df}\kappa}$. Intuitively, the one-pass transformation is derived by staging the program $\{\!|E|\!\}_{\mathsf{p}\kappa}$ [7].

By the definition of the translation, the two-level program $\{\!|E|\!\}_{\mathsf{df}\kappa}$ does not use the fixed-point operator. We can prove that the evaluation of such a term always terminates using a standard logical-relation argument (note that, with respect to the termination property, the code type behaves the same as a usual base type like int).[4] The question is how to ensure that the resulting term has the same behavior as the output of Plotkin's original transformation, $\{\!|E|\!\}_{\mathsf{p}\kappa}$. An intuitive argument is that erasing the annotations in $\{\!|E|\!\}_{\mathsf{df}\kappa}$ produces a term which is $\beta\eta$-equivalent to $\{\!|E|\!\}_{\mathsf{p}\kappa}$.

2.3 Semantic correctness of the generated code: erasure

The notion of *annotation erasure* formalizes the intuitive idea of erasing all the binding-time annotations, relates nPCF^2 to nPCF, and supports the general view of two-level programs as staged version of one-level programs.

Definition 2.2 (Erasure). *The (annotation) erasure of a* nPCF^2*-phrase is the* nPCF*-phrase given as follows.*
Types: $|\bigcirc\sigma| = \sigma$, $|\mathsf{b}| = \mathsf{b}$, $|\tau_1 \to \tau_2| = |\tau_1| \to |\tau_2|$.

[3] The call-by-name CPS transformation is studied in Appendix A.

[4] Less directly, we can also use the embedding translation introduced in Section 2.4 and its associated correctness theorem: The embedding of a term without fixed-point operators does not use the fixed-point operator either, and thus its evaluation terminates in the standard operational semantics.

a. Source syntax: the pure simply typed λ-calculus $\mathsf{v\Lambda}$

Types $\quad \sigma \ ::= \ \mathsf{b} \mid \sigma_1 \to \sigma_2$

Raw terms $\quad E \ ::= \ x \mid \lambda x.E \mid E_1\, E_2$

Typing judgment $\boxed{\mathsf{v\Lambda} \vdash \Delta \rhd E : \sigma}$ (omitted)

b. Plotkin's original transformation:

$$\boxed{\mathsf{v\Lambda} \vdash \Delta \rhd E : \sigma} \Rightarrow \boxed{\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}}.$$

Here, $K\sigma = (\sigma \to \mathsf{Ans}) \to \mathsf{Ans}$ for an answer type Ans.

Types: $\{\!|\mathsf{b}|\!\}_{\mathsf{p}\kappa} = \mathsf{b}$,

$\{\!|\sigma_1 \to \sigma_2|\!\}_{\mathsf{p}\kappa} = \{\!|\sigma_1|\!\}_{\mathsf{p}\kappa} \to K\{\!|\sigma_2|\!\}_{\mathsf{p}\kappa}$,

Terms: $\{\!|x|\!\}_{\mathsf{p}\kappa} = \lambda k.k\, x$,

$\{\!|\lambda x.E|\!\}_{\mathsf{p}\kappa} = \lambda k.k\, \lambda x.\{\!|E|\!\}_{\mathsf{p}\kappa}$,

$\{\!|E_1\, E_2|\!\}_{\mathsf{p}\kappa} = \lambda k.\{\!|E_1|\!\}_{\mathsf{p}\kappa} \lambda r_1.\{\!|E_2|\!\}_{\mathsf{p}\kappa} \lambda r_2.r_1\, r_2\, k.$

c. Danvy and Filinski's one-pass transformation:

$$\boxed{\mathsf{v\Lambda} \vdash \Delta \rhd E : \sigma} \implies \boxed{\mathsf{nPCF}^2 \vdash \bigcirc\{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd \{\!|E|\!\}_{\mathsf{df}^2\kappa} : K^{\bigcirc}(\bigcirc\{\!|\sigma|\!\}_{\mathsf{p}\kappa})}.$$

Here, $K^{\bigcirc}\tau = (\tau \to \bigcirc\mathsf{Ans}) \to \bigcirc\mathsf{Ans}$.

Terms:

$$\{\!|x|\!\}_{\mathsf{df}^2\kappa} = \lambda k.k\, x,$$

$$\{\!|\lambda x.E|\!\}_{\mathsf{df}^2\kappa} = \lambda k.k\, \underline{\lambda} x.\underline{\lambda} k'.\{\!|E|\!\}_{\mathsf{df}^2\kappa} \lambda m.k' \underline{@} m,$$

$$\{\!|E_1\, E_2|\!\}_{\mathsf{df}^2\kappa} = \lambda k.\{\!|E_1|\!\}_{\mathsf{df}^2\kappa} \lambda r_1.\{\!|E_2|\!\}_{\mathsf{df}^2\kappa} \lambda r_2.r_1 \underline{@} r_2 \underline{@}\, \underline{\lambda} a.k\, a.$$

The complete translation

$$\implies \boxed{\mathsf{nPCF}^2 \vdash \bigcirc\{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd \{\!|E|\!\}_{\mathsf{df}\kappa} : \bigcirc(K\{\!|\sigma|\!\}_{\mathsf{p}\kappa})}.$$

$$\{\!|E|\!\}_{\mathsf{df}\kappa} = \underline{\lambda} k.\{\!|E|\!\}_{\mathsf{df}^2\kappa} \lambda m.k \underline{@} m$$

Figure 4: Call-by-value CPS transformation

Terms: $|x| = x$, $|\$_{\mathsf{b}} E| = E$, $|\underline{d}| = d$, $|\underline{\lambda} x.E| = \lambda x.|E|$,
$|E_1 \underline{@} E_2| = |E_1|\,|E_2|$.
Erasure of the static term constructs is homomorphic (e.g., $|\mathbf{fix}\, E| = \mathbf{fix}\, |E|$, $|\lambda x.E| = \lambda x.|E|$*). If* $\mathsf{nPCF}^2 \vdash \Gamma \rhd E : \tau$*, then* $\mathsf{nPCF} \vdash |\Gamma| \rhd |E| : |\tau|$*. Finally, the object-level term represented by a code-typed value* $\mathcal{O}$ *is its erasure* $|\mathcal{O}|$*.*

The following theorem states that evaluation of two-level terms in nPCF^2 respects the nPCF-equality under erasure.

Theorem 2.3 (Annotation erasure). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF} \vdash \Delta \rhd |E| = |V| : |\tau|$*.*

Proof. By induction on $E \Downarrow V$. □

With Theorem 2.3, in order to show certain extensional properties of generated programs, it suffices to show them for the erasure of the original two-level program. As an example, we check the semantic correctness of the one-pass CPS transformation with respect to Plotkin's transformation.

Proposition 2.4 (Correctness of one-pass CPS). *If* $\mathsf{v}\Lambda \vdash \Delta \rhd E : \sigma$ *and* $\mathsf{nPCF}^2 \vdash \{\!|E|\!\}_{\mathsf{df}\kappa} \Downarrow \mathcal{O}$ *then* $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\mathcal{O}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$*.*

Proof. A simple induction on E establishes that $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\{\!|E|\!\}_{\mathsf{df}^2\kappa}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$, which has the immediate corollary that $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\{\!|E|\!\}_{\mathsf{df}\kappa}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$. We then apply Theorem 2.3. □

The proof of Proposition 2.4 embodies the basic pattern to establish semantic correctness based on annotation erasure. Although we are only interested in the extensional property of the generated code (which, we shall recall, is the erasure of the code-typed value $\mathcal{O}$ resulted from the evaluation), we need to recursively establish extensional properties (e.g., equal to specific one-level terms) for the erasures of all the sub-terms. Most of these sub-terms have higher types and do not generate code by themselves; for these subterms, Theorem 2.3 does not give any readily usable result about the semantics of code generation, since the theorem applies only to terms of code types. But since erasure is compositional, the extension of the sub-terms' erasures builds up to that of the complete program's erasure, for which Theorem 2.3 could deliver the result. It is worth noting the similarity between this process and the process of a proof based on a logical-relation argument.

2.4 Embedding nPCF^2 into a one-level language with a term type

Our goal is also to justify native implementations of code-generation algorithms. To this end, we want to embed the two-level language nPCF^2 in the one-level language nPCF^{Λ}, which is nPCF with object-level constants removed, and enriched with an inductive type Λ (the equational theory, correspondingly, is enriched with the congruence rules for the data constructors):

$$\begin{array}{rcl}\Lambda & = & \mathsf{VAR}\ \mathit{of}\ \mathsf{int} \mid \mathsf{LIT_b}\ \mathit{of}\ \mathsf{b} \mid \mathsf{CST}\ \mathit{of}\ \mathsf{const} \\ & \mid & \mathsf{LAM}\ \mathit{of}\ \mathsf{int} \times \Lambda \mid \mathsf{APP}\ \mathit{of}\ \Lambda \times \Lambda\end{array}$$

The type const provides a representation $\wr d \int$ for constants d—usually the string type suffices. Type Λ provides a representation for raw λ-terms whose variable names are of the form v_i for all natural numbers i: A value V of type Λ encodes the raw term $\mathcal{D}(V)$:

$$\begin{array}{l}\mathcal{D}(\mathsf{VAR}(i)) = v_i,\ \mathcal{D}(\mathsf{LIT_b}(\ell)) = \ell,\ \mathcal{D}(\mathsf{CST}(\wr d \int)) = d \\ \mathcal{D}(\mathsf{LAM}(i, e)) = \lambda v_i.\mathcal{D}(e),\ \mathcal{D}(\mathsf{APP}(e_1, e_2)) = \mathcal{D}(e_1)\,\mathcal{D}(e_2)\end{array}$$

The language nPCF^{Λ} has a standard, domain-theoretical CBN denotational semantics [34],[5] which interprets the types as follows:

$$[\![\mathsf{int}]\!] = \mathbf{Z}_\perp, [\![\mathsf{bool}]\!] = \mathbf{B}_\perp, [\![\Lambda]\!] = \mathbf{E}_\perp, [\![\sigma_1 \to \sigma_2]\!] = [\![\sigma_1]\!] \to [\![\sigma_2]\!]$$

where $\mathbf{Z}$, $\mathbf{B}$ and $\mathbf{E}$ are respectively the set of integers, the set of booleans, and the set of raw terms (i.e., the inductive set given as the smallest solution to the equation $X = \mathbf{Z} + \mathbf{Z} + \mathbf{B} + \mathbf{Cst} + \mathbf{Z} \times X + X \times X$). Without giving the detailed semantics (which can be found in Appendix B), we remark that (1) the equational theory is sound with respect to the denotational semantics: If $\mathsf{nPCF}^{\Lambda} \vdash \Delta \rhd E_1 = E_2 : \sigma$, then $[\![E_1]\!] = [\![E_2]\!]$, and (2) the evaluation function for closed terms of base types induced from the denotational semantics has (by its computational adequacy with respect to a environment-based (i.e., not substitution-based) call-by-name evaluation semantics where evaluation of the data constructors are strict; the proof of adequacy adapts the standard proof [46]) efficient implementations that do not perform capture-avoiding substitutions.

Definition 2.5 (Embedding of nPCF^{2} into nPCF^{Λ}: $\{\!|-|\!\}_{\mathsf{n}\epsilon}$).

$$\begin{array}{ll}\text{Types}: & \{\!|\bigcirc\sigma|\!\}_{\mathsf{n}\epsilon} = \mathsf{int} \to \Lambda, \{\!|\mathsf{b}|\!\}_{\mathsf{n}\epsilon} = \mathsf{b}, \\ & \{\!|\tau_1 \to \tau_2|\!\}_{\mathsf{n}\epsilon} = \{\!|\tau_1|\!\}_{\mathsf{n}\epsilon} \to \{\!|\tau_2|\!\}_{\mathsf{n}\epsilon} \\ \text{Terms}: & \{\!|\$_{\mathsf{b}} E|\!\}_{\mathsf{n}\epsilon} = \$_{\mathsf{b}}^{\mathsf{n}\epsilon}\{\!|E|\!\}_{\mathsf{n}\epsilon}, \{\!|\underline{d}|\!\}_{\mathsf{n}\epsilon} = \lambda i.\mathsf{CST}(\wr d \int), \\ & \{\!|\underline{\lambda} x.E|\!\}_{\mathsf{n}\epsilon} = \underline{\lambda}^{\mathsf{n}\epsilon}\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon}, \\ & \{\!|E_1 \underline{@} E_2|\!\}_{\mathsf{n}\epsilon} = \underline{@}^{\mathsf{n}\epsilon}\{\!|E_1|\!\}_{\mathsf{n}\epsilon}\{\!|E_2|\!\}_{\mathsf{n}\epsilon}\end{array}$$

where we use the following terms:

$$\begin{array}{l}\$_{\mathsf{b}}^{\mathsf{n}\epsilon} \equiv \lambda l.\lambda i.\mathsf{LIT_b}(l), \\ \underline{\lambda}^{\mathsf{n}\epsilon} \equiv \lambda f.\lambda i.\mathsf{LAM}(i, f(\lambda i'.\mathsf{VAR}(i))(i+1)), \\ \underline{@}^{\mathsf{n}\epsilon} \equiv \lambda m.\lambda n.\lambda i.\mathsf{APP}(mi, ni).\end{array}$$

Static constructs are translated homomorphically.

[5] This is different from a lazy CBN semantics [54], which models Haskell-like languages where higher-order types are observable; there, function spaces are lifted, and the η-rule does not hold.

The three terms used in the embedding translation are nPCF^{Λ}-terms themselves, and are kept as-is in the result of the translation. For instance, $\{\!|f\underline{@}x|\!\}_{\mathsf{n}\epsilon}$ is the term $\underline{@}^{\mathsf{n}\epsilon}fx \equiv (\lambda m.\lambda n.\lambda i.\mathsf{APP}(mi, ni))fx$, not the simplified term $\lambda i.fi, xi$. This is crucial for the validity of the following substitution lemma (Lemma 2.6); moreover, this also models the actual implementation, where the dynamic constructs are provided as combinators in the implementation language nPCF^{Λ}.

The translation uses a de Bruijn-level encoding for generating variable bindings. Furthermore, a dynamic λ-abstraction is translated using a static λ-abstraction[6] and thus the two terms have the same binding behavior—a fact reflected in the following substitution lemma.

Lemma 2.6 (Substitution lemma for $\{\!|-|\!\}_{\mathsf{n}\epsilon}$). *If* $\mathsf{nPCF}^{2} \vdash \Gamma, x : \tau' \rhd E : \tau$ *and* $\mathsf{nPCF}^{2} \vdash \Gamma \rhd E' : \tau'$, *then* $\{\!|E\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \sim_{\alpha} \{\!|E|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\}$.

We shall establish that the embedding translation preserves the behavior of closed terms of the code type, $\bigcirc\sigma$ in nPCF^{2} and Λ in nPCF^{Λ}.

Lemma 2.7 (Evaluation preserves translation). *If* $\mathsf{nPCF}^{2} \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^{2} \vdash E \Downarrow V$, *then* $\mathsf{nPCF}^{\Lambda} \vdash \{\!|\bigcirc\Delta|\!\}_{\mathsf{n}\epsilon} \rhd \{\!|E|\!\}_{\mathsf{n}\epsilon} = \{\!|V|\!\}_{\mathsf{n}\epsilon} : \{\!|\tau|\!\}_{\mathsf{n}\epsilon}$.

Proof. By induction on $\mathsf{nPCF}^{2} \vdash E \Downarrow V$. For $E \equiv E_1\, E_2$, we use Lemma 2.6. ☐

Lemma 2.8 (Translation of code-typed value). *If* $\mathsf{nPCF}^{2} \vdash v_1{:}\bigcirc\sigma_1, \ldots, v_n{:} \bigcirc\sigma_n \rhd \mathcal{O} : \bigcirc\sigma$, *then there is a value* $t : \Lambda$ *such that*

- $\mathsf{nPCF}^{\Lambda} \vdash\ \rhd (\{\!|\mathcal{O}|\!\}_{\mathsf{n}\epsilon}(n+1))\{\lambda i.\mathsf{VAR}(1)/v_1, \ldots, \lambda i.\mathsf{VAR}(n)/v_n\} = t : \Lambda$,
- $\mathsf{nPCF} \vdash v_1 : \sigma_1, \ldots, v_n : \sigma_n \rhd \mathcal{D}(t) : \sigma$, *and*
- $|\mathcal{O}| \sim_{\alpha} \mathcal{D}(t)$.

Proof. By induction on the size of term $\mathcal{O}$. For the case $\mathcal{O} \equiv \underline{\lambda}x.\mathcal{O}_1$, we use induction hypothesis on the term $\mathcal{O}_1\{v_{n+1}/x\}$. ☐

Lemma 2.9 (Computational adequacy). *If* $\mathsf{nPCF}^{2} \vdash \rhd E : \bigcirc\sigma$, *and there is a* nPCF^{Λ}*-value* $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$, *then* $\exists\mathcal{O}.E \Downarrow \mathcal{O}$.

Proof. (Sketch) We use a Kripke logical relation between nPCF^{2}-terms and the standard denotational semantics of nPCF^{Λ}, which relates a term E and the denotation of $\{\!|E|\!\}_{\mathsf{n}\epsilon}$. The definition of the logical relation at the type $\bigcirc\sigma$ implies the conclusion. ☐

Theorem 2.10 (Correctness of embedding). *If* $\mathsf{nPCF}^{2} \vdash \rhd E : \bigcirc\sigma$, *then the following statements are equivalent.*
(a) There is a value $\mathcal{O} : \bigcirc\sigma$ *such that* $\mathsf{nPCF}^{2} \vdash E \Downarrow \mathcal{O}$.

[6]This is an instance of higher-order abstract syntax [44]. It might come as a surprise that we use both higher-order abstract syntax and de Bruijn levels. In fact, they serve two different but related functions: higher-order abstract syntax makes the object-level capturing-behavior consistent with the meta-level capturing-behavior, and the de Bruijn levels are used to generate the concrete names of the object terms.

(b) There is a value $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$.
When these statements hold, we further have that
(c) $\mathsf{nPCF} \vdash \rhd \mathcal{D}(t) : \sigma$ *and* $|\mathcal{O}| \sim_\alpha \mathcal{D}(t)$.

Proof. (a) $\Rightarrow$ (b),(c): We combine Lemmas 2.7 and 2.8 to show the existence of value $t : \Lambda$ such that (c) holds and $\mathsf{nPCF}^\Lambda \vdash \rhd \{\!|E|\!\}_{\mathsf{n}\epsilon}(1) = t : \Lambda$, which implies (b) by the soundness of the type theory.

(b) $\Rightarrow$ (a),(c): By Lemma 2.9, we have (a); by (a) $\Rightarrow$ (c), we have a value $t' : \Lambda$ that satisfies (c). It is easy to show that $t \equiv t'$. $\square$

The embedding provides a *native* implementation of nPCF^2 in nPCF^Λ: Static constructs are translated to themselves and dynamic constructs can be defined as functions. Explicit substitution in the operational semantics has been simulated using de Bruijn-style name generation through the translation. The code generation in the implementation is one-pass, in that code is only generated once, without further traversal over it, as in a substitution-based implementation. Furthermore, native embeddings exploit the potentially efficient implementation of the one-level language, and they also offer users the flexibility to use extra syntactic constructs in the one-level language—as long as these constructs are semantically equivalent to terms in the proved part.

2.5 Example: call-by-name type-directed partial evaluation

We now turn to a bigger example: Type-Directed Partial Evaluation (TDPE) [6]. Following Filinski's formalization [12], we describe TDPE as a native normalization process for fully dynamic terms (i.e., terms whose types are built solely from dynamic base types) in the somewhat different two-level language $\mathsf{nPCF}^{\mathsf{tdpe}}$. Here, by a native normalization process, we mean an normalization algorithm that is implemented through a native embedding from $\mathsf{nPCF}^{\mathsf{tdpe}}$ into the implementation language.

The syntax of $\mathsf{nPCF}^{\mathsf{tdpe}}$ is displayed in Figure 5a. The language $\mathsf{nPCF}^{\mathsf{tdpe}}$ differs from nPCF^2 in that only base types are binding-time annotated as static (b) or dynamic ($\mathsf{b}^\mathfrak{d}$, instead of $\bigcirc\mathsf{b}$, for clarity), and the language does *not* have any dynamic type constructors (like the dynamic function type in nPCF^2). Apart from lifted literals, dynamic constants $d^\mathfrak{d}$ are the only form of term construction that introduces dynamic types. Their types, written in the form $\sigma^\mathfrak{d}$, are the fully dynamic counterpart of the constants' type σ in the object language: For example, for the object-level constant $\mathsf{eq} : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, the corresponding dynamic constant is $\mathsf{eq}^\mathfrak{d} : \mathsf{int}^\mathfrak{d} \to \mathsf{int}^\mathfrak{d} \to \mathsf{bool}^\mathfrak{d}$; consequently, for what we write $\underline{\mathsf{eq}}\underline{@}(\$_{\mathsf{int}}(1+2)) : \bigcirc(\mathsf{int}{\to}\mathsf{bool})$ in nPCF^2, we write $\mathsf{eq}^\mathfrak{d}\ (\$_{\mathsf{int}}(1+2)) : \mathsf{int}^\mathfrak{d}{\to}\mathsf{bool}^\mathfrak{d}$ in $\mathsf{nPCF}^{\mathsf{tdpe}}$. Let us stress that there is no binding-time annotation for applications here.

The semantics is described through a *standard instantiation*[7] into the one-

[7] An instantiation is a homomorphic syntactic translation. It is specified by a substitution from the base types to types and from constants to terms.

a. The language of CBN TDPE: $\mathsf{nPCF}^{\mathsf{tdpe}}$

Type $\quad \varphi ::= \mathsf{b} \mid \mathsf{b}^{\mathfrak{d}} \mid \varphi_1 \to \varphi_2$

Raw terms $\quad E ::= x \mid \ell \mid \lambda x.E \mid E_1\, E_2 \mid \mathbf{fix}\, E_1 \mid \mathbf{if}\, E_1\, E_2\, E_3 \mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid d^{\mathfrak{d}}$

Typing judgment: $\boxed{\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi}$

Typing rules: same as those of nPCF^2, with the dynamic ones replaced by

$$[\mathit{cst}^{\mathfrak{d}}]\, \frac{Sg(d) = \sigma}{\Gamma \rhd d^{\mathfrak{d}} : \sigma^{\mathfrak{d}}} \qquad [\mathit{lift}^{\mathfrak{d}}]\, \frac{\Gamma \rhd E : \mathsf{b}}{\Gamma \rhd \$_{\mathsf{b}} E : \mathsf{b}^{\mathfrak{d}}},$$

where $\sigma^{\mathfrak{d}} \triangleq \sigma\{\mathsf{b}^{\mathfrak{d}}/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$, i.e., fully dynamic counterpart of the type σ.

b. Standard instantiation (TDPE-erasure)

$\boxed{\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi} \Longrightarrow \boxed{\mathsf{nPCF} \vdash |\Gamma| \rhd |E| : |\varphi|}$

$$|\mathsf{b}^{\mathfrak{d}}| = \mathsf{b}; \qquad |\$_{\mathsf{b}} E| = |E|, |d^{\mathfrak{d}}| = d$$

c. Extraction functions $\downarrow^{\sigma}$ and $\uparrow_{\sigma}$:

We write $\sigma^{\bigcirc}$ for $\sigma\{\bigcirc\mathsf{b}/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$.

$$\begin{cases} \mathsf{nPCF}^2 \vdash \rhd\downarrow^{\sigma} : \sigma^{\bigcirc} \to \bigcirc\sigma \\ \downarrow^{\mathsf{b}} = \lambda x.x \\ \downarrow^{\sigma_1 \to \sigma_2} = \lambda f.\underline{\lambda} x.\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x)) \end{cases}$$

$$\begin{cases} \mathsf{nPCF}^2 \vdash \rhd\uparrow_{\sigma} : \bigcirc\sigma \to \sigma^{\bigcirc} \\ \uparrow_{\mathsf{b}} = \lambda x.x \\ \uparrow_{\sigma_1 \to \sigma_2} = \lambda e.\lambda x.\uparrow_{\sigma_2}(e\,\underline{@}\,(\downarrow^{\sigma_1} x)) \end{cases}$$

d. Residualizing instantiation

$\boxed{\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi} \Rightarrow \boxed{\mathsf{nPCF}^2 \vdash \{\!|\Gamma|\!\}_{\mathsf{ri}} \rhd \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}_{\mathsf{ri}}}$

$$\{\!|\mathsf{b}^{\mathfrak{d}}|\!\}_{\mathsf{ri}} = \bigcirc\mathsf{b}; \qquad \{\!|\$_{\mathsf{b}} E|\!\}_{\mathsf{ri}} = \$_{\mathsf{b}}\{\!|E|\!\}_{\mathsf{ri}}, \{\!|d^{\mathfrak{d}} : \sigma^{\mathfrak{d}}|\!\}_{\mathsf{ri}} = \uparrow_{\sigma}\underline{d}$$

e. The static normalization function NF is defined on closed terms E of fully dynamic types $\sigma^{\mathfrak{d}}$:

$$NF(E : \sigma^{\mathfrak{d}}) = \downarrow^{\sigma}\{\!|E|\!\}_{\mathsf{ri}} : \bigcirc\sigma$$

Figure 5: Call-by-name type-directed partial evaluation

level language nPCF (Figure 5b), which amounts to erasing all the annotations; thus we overload the notation of erasure here.

Normalizing a closed $\mathsf{nPCF}^{\mathsf{tdpe}}$-term E of fully dynamic type $\sigma^{\mathfrak{d}}$ amounts to finding a nPCF-term $E' : \sigma$ in long $\beta\eta$-normal form (fully η-expanded terms with no β-redexes; see Section 2.6 for detail) such that $\mathsf{nPCF} \vdash\ \rhd |E| = E' : \sigma$. For example, normalizing the term $\mathsf{eq}^{\mathfrak{d}}\ (\$_{\mathsf{int}}(1+2))$ should produce the object term $\lambda x.\mathsf{eq}\,3\,x$. As in Filinski's treatment, this notion of normalization leaves the dynamic constants uninterpreted—E and E' need to be the same under all interpretations of constants, since there are no equations for dynamic constants.

The TDPE algorithm, formulated in nPCF^2, is shown in Figure 5c-e. It finds the normal form of a $\mathsf{nPCF}^{\mathsf{tdpe}}$-term $E : \sigma^{\mathfrak{d}}$ by applying a type-indexed extraction function $\downarrow^{\sigma}$ ("reification") on a particular instantiation, called the *residualizing instantiation* $\{\!|E|\!\}_{\mathsf{ri}}$, of term E in the language nPCF^2. Being an instantiation, which maps static constructs to themselves, $\{\!|\cdot|\!\}_{\mathsf{ri}}$ makes the TDPE algorithm natively implementable in nPCF^{Λ} through the embedding $\{\!|-|\!\}_{\mathsf{n}\epsilon}$ of Section 2.4. Indeed, the composition of $\{\!|-|\!\}_{\mathsf{ri}}$ and the embedding $\{\!|-|\!\}_{\mathsf{n}\epsilon}$ is essentially the same as Filinski's direct formulation in the one-level language.

We first use the erasure argument to show that the result term of TDPE is semantically correct, i.e., that the term generated by running $NF(E)$ has the same semantics as the standard instantiation $|E|$ of E.

Lemma 2.11. *For all types σ, $\mathsf{nPCF} \vdash\ \rhd |\downarrow^{\sigma}| = \lambda x.x : \sigma \to \sigma$ and $\mathsf{nPCF} \vdash\ \rhd |\uparrow_{\sigma}| = \lambda x.x : \sigma \to \sigma$.*

The lemma captures the intuition of TDPE as two-level η-expansion, as Danvy stated in his initial presentation of TDPE [5].

Theorem 2.12 (Semantic correctness of TDPE). *If $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$ and $\mathsf{nPCF}^2 \vdash NF(E) \Downarrow \mathcal{O}$, then $\mathsf{nPCF} \vdash\ \rhd |\mathcal{O}| = |E| : \sigma$.*

Proof. A simple induction on E establishes that $\mathsf{nPCF} \vdash\ \rhd |\{\!|E|\!\}_{\mathsf{ri}}| = |E| : \sigma$, which has the immediate corollary that $\mathsf{nPCF} \vdash\ \rhd |NF(E)| = |E| : \sigma$. We then apply Theorem 2.3. □

2.6 Syntactic correctness of the generated code: type preservation

Semantic correctness of the generated terms does not give much syntactic guarantee of the generated terms, but using the standard type preservation (Theorem 2.1), we can already infer some intensional properties about the output of TDPE: It does not contain static constructs, and it is typable in nPCF. Furthermore, a quick inspection of the TDPE algorithm reveals that it will never construct a β-redex in the output—since there is no way to pass a dynamic λ-abstraction to the $\uparrow$ function. Indeed, an ad-hoc native implementation can be easily refined to express this constraint by changing the term type Λ. To capture that the output is fully η-expanded by typing, however, appears to require dependent types for the term representation.[8]

[8]On the other hand, through some extra reasoning on the way the two-level program is written, it is possible to prove that the output is fully η-expanded in such a setting, as done by Danvy and Rhiger recently [9].

To show that the output of TDPE is always in long $\beta\eta$-normal form, i.e., typable according to the rules in Figure 6 (directly taken from Filinski [12]), we can take inspiration from the evaluation of nPCF^2-terms of type $\bigcirc\sigma$. Type preservation shows that evaluating these terms always yields a value of type $\bigcirc\sigma$, which corresponds to a well-typed nPCF-term. Similarly, to show that evaluating $NF(E)$ always yields long $\beta\eta$-terms, we can refine the dynamic typing rules of nPCF^2, so that values of code type correspond to terms in long $\beta\eta$-normal form, and then we show that (1) evaluation preserves typing in the new type system; and (2) the term $NF(E)$ is always typable in this new type system.

The two-level language with dynamic typing rules refined according to the rules for long $\beta\eta$-normal forms is shown in Figure 7. Briefly, we attach the sort of the judgment, atomic *at* or normal form *nf*, with the code type, and add another code type $\bigcirc^{var}(-)$ for variables in the context. This way, evaluation of static β-redexes will not substitute the wrong sort of syntactic phrase and introduce ill-formed code. The type system is a refinement of the original type system in the sense that all the new dynamic typing rules are derivable in the original system, if we ignore the new "refinement" tags (*at*, *nf*, *var*), and hence any term typable in the new type system is trivially typable in the original one.

$$\frac{\Delta \rhd^{at} E : \mathsf{b}}{\Delta \rhd^{nf} E : \mathsf{b}} \qquad \frac{\Delta, x : \sigma_1 \rhd^{nf} E : \sigma_2}{\Delta \rhd^{nf} \lambda x.E : \sigma_1 \to \sigma_2} \qquad \frac{\ell \in \mathbb{L}(\mathsf{b})}{\Delta \rhd^{at} \ell : \mathsf{b}}$$

$$\frac{Sg(d) = \sigma}{\Delta \rhd^{at} d : \sigma} \qquad \frac{x{:}\,\sigma \in \Delta}{\Delta \rhd^{at} x : \sigma} \qquad \frac{\Delta \rhd^{at} E_1 : \sigma_2 \to \sigma \quad \Delta \rhd^{nf} E_2 : \sigma_2}{\Delta \rhd^{at} E_1\, E_2 : \sigma}$$

Figure 6: Inference rules for terms in long $\beta\eta$-normal form

Types $\tau ::= \mathsf{b} \mid \bigcirc^{var}(\sigma) \mid \bigcirc^{nf}(\sigma) \mid \bigcirc^{at}(\sigma)$

Typing Judgment $\boxed{\mathsf{nPCF}^2 \vdash \Gamma \blacktriangleright E : \tau}$

(Static) same as the static rules for $\mathsf{nPCF}^2 \vdash \Gamma \rhd E : \tau$

(Dynamic)

$$\frac{\Gamma \blacktriangleright E : \bigcirc^{at}(\mathsf{b})}{\Gamma \blacktriangleright E : \bigcirc^{nf}(\mathsf{b})} \qquad \frac{\Gamma, x : \bigcirc^{var}(\sigma_1) \blacktriangleright E : \bigcirc^{nf}(\sigma_2)}{\Gamma \blacktriangleright \underline{\lambda} x.E : \bigcirc^{nf}(\sigma_1 \to \sigma_2)} \qquad \frac{\Gamma \blacktriangleright E : \mathsf{b}}{\Gamma \blacktriangleright \$_{\mathsf{b}} E : \bigcirc^{at}(\mathsf{b})}$$

$$\frac{Sg(d) = \sigma}{\Gamma \blacktriangleright \underline{d} : \bigcirc^{at}(\sigma)} \qquad \frac{\Gamma \blacktriangleright E : \bigcirc^{var}(\sigma)}{\Gamma \blacktriangleright E : \bigcirc^{at}(\sigma)}$$

$$\frac{\Gamma \blacktriangleright E_1 : \bigcirc^{at}(\sigma_2 \to \sigma) \quad \Gamma \blacktriangleright E_2 : \bigcirc^{nf}(\sigma_2)}{\Gamma \blacktriangleright E_1 \underline{@} E_2 : \bigcirc^{at}(\sigma)}$$

Figure 7: nPCF^2-terms that generate code in long $\beta\eta$-normal form

Theorem 2.13 (Refined type preservation). *If* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \tau$*.*

Theorem 2.14 (Normal-form code types). *If V is an* $\mathsf{nPCF^2}$*-value (Figure 2), then*
(1) if $\mathsf{nPCF^2} \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \bigcirc^{at}(\sigma)$*, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^{at} |\mathcal{O}| : \sigma$*;*
(2) if $\mathsf{nPCF^2} \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \bigcirc^{nf}(\sigma)$*, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^{nf} |\mathcal{O}| : \sigma$*.*

For our example, we are left to check that the TDPE algorithm can be typed with normal-form types in this calculus.

Lemma 2.15. *(1) The extraction functions (Figure 5c) have the following normal-form types (writing* $\sigma^{\bigcirc\mathsf{nf}}$ *for* $\sigma\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*).*

$$\blacktriangleright \downarrow^{\sigma} : \sigma^{\bigcirc\mathsf{nf}} \to \bigcirc^{nf}(\sigma), \blacktriangleright \uparrow_{\sigma} : \bigcirc^{at}(\sigma) \to \sigma^{\bigcirc\mathsf{nf}}.$$

(2) If $\mathsf{nPCF^{tdpe}} \vdash \Gamma \rhd E : \varphi$*, then* $\mathsf{nPCF^2} \vdash \{\!|\Gamma|\!\}^{\mathsf{nf}}_{\mathsf{ri}} \blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}^{\mathsf{nf}}_{\mathsf{ri}}$*, where* $\{\!|\varphi|\!\}^{\mathsf{nf}}_{\mathsf{ri}} = \varphi\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b}^{\mathfrak{d}} : \mathsf{b} \in \mathbb{B}\}$

Theorem 2.16. *If* $\mathsf{nPCF^{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$*, then* $\mathsf{nPCF^2} \vdash\blacktriangleright NF(E) : \bigcirc^{nf}(\sigma)$*.*

Corollary 2.17 (Syntactic correctness of TDPE). *For* $\mathsf{nPCF^{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$*, if* $\mathsf{nPCF^2} \vdash NF(E) \Downarrow V$*, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\mathsf{nPCF} \vdash \Delta \rhd^{nf} |\mathcal{O}| : \sigma$*.*

It appears possible to give a general treatment for refining the dynamic part of the typing judgment, and establish once and for all that such typing judgments come equipped with the refined type preservation, using Plotkin's notion of binding signature to specify the syntax of the object language [14, 47]. However, since the object language is typed, we need to use a binding signature with dependent types, which could be complicated. We therefore leave this general treatment to a future work.

3 The general framework

In Section 2, we have seen how several properties of the language $\mathsf{nPCF^2}$ aid in reasoning about code-generation programs and their native implementation in one-level languages. Before moving on, let us identify the general conceptual structure underlying the development.

The aim is to facilitate writing and reasoning about code-generation algorithms through the support of a two-level language over a specific object language. Following the code-type view, we do not insist, from the outset, that the static language and the dynamic language should be the same. But to accommodate the staging view, we collapse the two-level language, say $\mathsf{L^2}$ (e.g., $\mathsf{nPCF^2}$), into a corresponding one-level language, say L (e.g., nPCF), for which a more conventional axiomatic semantics (an equational theory in this article) can be used for reasoning.

Using a high-level operational semantics of $\mathsf{L^2}$, we identify and prove properties of $\mathsf{L^2}$ that support the following two proof obligations:

Syntactic correctness of the generated code, i.e., it satisfies certain intensional, syntactic constraints, specified as typing rules I. We show that the code-type view is fruitful here: to start with, the values of a code type already represent well-typed terms in the object language (which can be modeled as a free binding algebra [14]). By establishing the **type preservation** theorem for the type system, we further have that code-typed programs generate only well-typed terms.

Similarly, for specific applications that require generated code to be I-typable, we can refine the code type, much like we do with an algebraic data type, by changing the dynamic typing rules according to I, so that code-typed values correspond only to I-typable terms. Subsequently, a **refined type preservation** theorem further ensures that the code-typed programs typable in the refined type system generate only I-typable terms. The original proof obligation is thus reduced to showing that the original two-level term type-checks in the refined type system.

Semantic correctness of the generated code, i.e., it satisfies a certain extensional property P. We use the **annotation-erasure** property from the staging view. Formulated using the equational theory of the object language L, this property states that if a two-level program E generates a term g, then g and the erasure $|E|$ of E must be equivalent: $\mathsf{L} \vdash g = |E|$. The original proof obligation is reduced to showing that P holds for $|E|$.

Implementation efficiency of the code-generation program, i.e., it can be efficiently implemented in a conventional one-level language, without actually carrying out symbolic reduction. By establishing a native embedding of L^2 into a conventional one-level language, we equip the two-level language with an efficient implementation that exploits the potentially optimized implementation of the one-level language.

In Section 2, the call-by-name, effect-free setting of nPCF^2 has made the proofs of the aforementioned properties relatively easy. It is reasonable to ask how applicable this technique is in other, probably more "realistic" settings. In the next section, we offer some initial positive answer: These properties should be taken into account in the design of new two-level languages to facilitate simple reasoning.

4 The call-by-value two-level language vPCF^2

In this section we design a two-level language vPCF^2 with Moggi's computational λ-calculus λ_c [35] as its object language in such a way that the language has the desired properties that we identified in Section 3 (Section 4.1). These properties are used to give a clean account of call-by-value TDPE (Section 4.3).

4.1 Design considerations

Since we aim at some form of erasure argument, the static part of the language should have a semantics compatible with the object language. We can consider a call-by-value (CBV) language for the static part and term construction for the dynamic part. Can we use the standard evaluation semantics of CBV languages for the static part as well?

The problematic rule is that of static function applications:

$$\frac{E_1 \Downarrow \lambda x.E' \quad E_2 \Downarrow V' \quad E'\{V'/x\} \Downarrow V}{E_1\, E_2 \Downarrow V}.$$

Even though the argument is evaluated to a value V', its erasure might still be an effectful computation (I/O, side effect, etc.): This happens when the argument E_2 is of some code type, so that V' is of the form $\mathcal{O}$. The evaluation rule then becomes unsound with respect to its erasure in the λ_c-theory. For example, let $E_2 \triangleq \underline{\mathsf{print}}\underline{@}(\$_{\mathsf{int}}(2+2))$, where $\underline{\mathsf{print}} : \bigcirc(\mathsf{int} \to \mathsf{bool})$ is a dynamic constant. Then the code generated by the program $(\lambda x.\underline{\mathbf{let}}\ y \Leftarrow x\ \underline{\mathbf{in}}\ x)\, E_2$ after erasure would be $\mathbf{let}\ y \Leftarrow (\mathsf{print}\, 4)\ \mathbf{in}\ (\mathsf{print}\, 4)$, which incorrectly duplicates the computation $\mathsf{print}\, 4$.

This problem can be solved by using the canonical technique of let-insertion in partial evaluation [3]: When V' is of the form $\mathcal{O}$ that represents an effectful computation, a let-binding $x = \mathcal{O}$ will be inserted at the enclosing residual binding (λ-abstraction or let-binding) and the variable x will be used in place of $\mathcal{O}$. But since we want vPCF^2 to be natively implementable in a conventional language, we should not change the evaluation rule for static applications. Our solution is to introduce a new code type $\text{ⓥ}\sigma$ whose values correspond to syntactical values, i.e., literals, variables, λ-abstractions, and constants. Only terms of such code type can appear at the argument position of an application. The usual code type, now denoted by $\text{ⓔ}\sigma$ to indicate possible computational effects, can be coerced into type $\text{ⓥ}\sigma$ with a "trivialization" operator $\#$, which performs let-insertion.

4.2 Syntax, semantics, and properties

The syntax and evaluation semantics of vPCF^2 are shown in Figure 8 and 9. Again, the languages are parameterized over a signature of typed constants. Due to the differences between call-by-name and call-by-value languages, the type of many important constants might differ: For example, for the conditional construct, we should have object-level constants $\mathsf{if}_\sigma : \mathsf{bool} \to (\mathsf{unit} \to \sigma) \to (\mathsf{unit}{\to}\sigma)$ (where unit is the standard unit type, which we omit from our language specification for the sake of brevity).

Note that the type θ of a function argument must be "substitution-safe", i.e., it cannot take the form $\text{ⓔ}\sigma$. The corresponding one-level language vPCF is an instance of the λ_c-calculus: Its syntax is the same as nPCF of Figure 3, except for an extra let-construct of the form $\mathbf{let}\ x \Leftarrow E_1\ \mathbf{in}\ E_2$ with the standard typing rule; its equational theory, an instance of Moggi's λ_c, includes β_v and η_v (the value-restricted version of the usual β and η rule), and conversion rules that commute $\mathbf{let}$ and other constructs.

In the evaluation semantics of vPCF^2, the accumulated bindings B are explicit; furthermore, the dynamic environment Δ is necessary, because the gen-

Types	$\tau ::= \theta \mid ⓔ\sigma$	
	$\theta ::= \mathsf{b} \mid ⓥ\sigma \mid \theta \to \tau$	(substitution-safe types)
	$\sigma ::= \mathsf{b} \mid \sigma_1 \to \sigma_2$	(object-code types)
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : \theta$	
Raw terms	$E ::= \ell \mid x \mid \lambda x.E \mid E_1\, E_2 \mid \mathbf{fix}\, E \mid \mathbf{if}\, E_1\, E_2\, E_3$ $\mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid \underline{\lambda} x.E \mid E_1 \underline{@} E_2 \mid \underline{d}$ $\mid \underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 \mid \#E$	

Typing Judgment $\boxed{\mathsf{vPCF}^2 \vdash \Gamma \rhd E : \tau}$

(Static)

$$[\mathit{lit}]\,\frac{\ell \in \mathbb{L}(\mathsf{b})}{\Gamma \rhd \ell : \mathsf{b}} \qquad [\mathit{var}]\,\frac{x{:}\,\theta \in \Gamma}{\Gamma \rhd x : \theta} \qquad [\mathit{lam}]\,\frac{\Gamma, x{:}\,\theta_1 \rhd E : \tau_2}{\Gamma \rhd \lambda x.E : \theta_1 \to \tau_2}$$

$$[\mathit{app}]\,\frac{\Gamma \rhd E_1 : \theta_2 \to \tau \quad \Gamma \rhd E_2 : \theta_2}{\Gamma \rhd E_1\, E_2 : \tau} \qquad [\mathit{fix}]\,\frac{\Gamma \rhd E : (\theta_1 \to \tau_2) \to (\theta_1 \to \tau_2)}{\Gamma \rhd \mathbf{fix}\, E : \theta_1 \to \tau_2}$$

$$[\mathit{if}]\,\frac{\Gamma \rhd E_1 : \mathsf{bool} \quad \Gamma \rhd E_2 : \tau \quad \Gamma \rhd E_3 : \tau}{\Gamma \rhd \mathbf{if}\, E_1\, E_2\, E_3 : \tau}$$

$$[\mathit{bop}]\,\frac{\Gamma \rhd E_1 : \mathsf{b}_1 \quad \Gamma \rhd E_2 : \mathsf{b}_2}{\Gamma \rhd E_1 \otimes E_2 : \mathsf{b}}(\otimes : \mathsf{b}_1 \times \mathsf{b}_2 \to \mathsf{b})$$

(Dynamic)

$$[\underline{\mathit{lift}}]\,\frac{\Gamma \rhd E : \mathsf{b}}{\Gamma \rhd \$_{\mathsf{b}} E : ⓥ\mathsf{b}} \qquad [\underline{\mathit{cst}}]\,\frac{Sg(d) = \sigma}{\Gamma \rhd \underline{d} : ⓥ\sigma} \qquad [\underline{\mathit{lam}}]\,\frac{\Gamma, x{:}ⓥ\sigma_1 \rhd E : ⓔ\sigma_2}{\Gamma \rhd \underline{\lambda} x.E : ⓥ(\sigma_1 \to \sigma_2)}$$

$$[\underline{\mathit{app}}]\,\frac{\Gamma \rhd E_1 : ⓔ(\sigma_2 \to \sigma) \quad \Gamma \rhd E_2 : ⓔ\sigma_2}{\Gamma \rhd E_1 \underline{@} E_2 : ⓔ\sigma} \qquad [\underline{\mathit{val}}]\,\frac{\Gamma \rhd E : ⓥ\sigma}{\Gamma \rhd E : ⓔ\sigma}$$

$$[\underline{\mathit{let}}]\,\frac{\Gamma, x{:}ⓥ\sigma_1 \rhd E_2 : ⓔ\sigma_2 \quad \Gamma \rhd E_1 : ⓔ\sigma_1}{\Gamma \rhd \underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 : ⓔ\sigma_2} \qquad [\underline{\mathit{triv}}]\,\frac{\Gamma \rhd E : ⓔ\sigma}{\Gamma \rhd \#E : ⓥ\sigma}$$

Figure 8: The type system of vPCF²

eration of new names is explicit in the semantics. The only rules that involve explicit manipulation of the bindings are those for the evaluation of dynamic lambda abstraction and dynamic let-expression (both of which initialize a local accumulator in the beginning, and insert the accumulated bindings at the end), and for the trivialization operator # (which inserts a binding to the accumulator).

In the following, by an abuse of notation, B also also stands for its own context part.

Let us examine the desired properties.

Values $V ::= \ell \mid \lambda x.E \mid \mathcal{O}$

$\mathcal{O} ::= \$_{\mathsf{b}}\ell \mid x \mid \underline{\lambda} x.\mathcal{O} \mid \mathcal{O}_1 \underline{@} \mathcal{O}_2 \mid \underline{d} \mid \underline{\mathbf{let}}\ x \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ \mathcal{O}_2$

Bindings $B ::= \cdot \mid B, x : \sigma = \mathcal{O}$

Judgment form $\boxed{\mathsf{vPCF}^{2} \vdash \Delta \rhd [B]E \Downarrow [B']V}$

We use the following abbreviations.

$$\frac{E_1 \Downarrow V_1 \quad \cdots \quad E_n \Downarrow V_n}{E \Downarrow V} \equiv \frac{\Delta \rhd [B_1]E_1 \Downarrow [B_2]V_1 \quad \cdots \quad \Delta \rhd [B_n]E_n \Downarrow [B_{n+1}]V_n}{\Delta \rhd [B_1]E \Downarrow [B_{n+1}]V}$$

$$\underline{\mathbf{let}}^*\ x_1{:}\sigma_1{=}\mathcal{O}_1, \cdots, x_n{:}\sigma_n{=}\mathcal{O}_n\ \underline{\mathbf{in}}\ \mathcal{O} \equiv \underline{\mathbf{let}}\ x_1 \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ (\cdots(\underline{\mathbf{let}}\ x_n \Leftarrow \mathcal{O}_n\ \underline{\mathbf{in}}\ \mathcal{O})\cdots)$$

(Static)

$$[\mathit{lit}]\ \frac{}{\ell \Downarrow \ell} \qquad [\mathit{lam}]\ \frac{}{\lambda x.E \Downarrow \lambda x.E} \qquad [\mathit{app}]\ \frac{E_1 \Downarrow \lambda x.E' \quad E_2 \Downarrow V' \quad E'\{V'/x\} \Downarrow V}{E_1\, E_2 \Downarrow V}$$

$$[\mathit{fix}]\ \frac{E \Downarrow \lambda x.E' \quad E'\{\mathbf{fix}\,(\lambda x.E')/x\} \Downarrow V}{\mathbf{fix}\, E \Downarrow V} \qquad [\mathit{if}\text{-}\mathsf{tt}]\ \frac{E_1 \Downarrow \mathsf{tt} \quad E_2 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V}$$

$$[\mathit{if}\text{-}\mathsf{ff}]\ \frac{E_1 \Downarrow \mathsf{ff} \quad E_3 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V} \qquad [\otimes]\ \frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{E_1 \otimes E_2 \Downarrow V}(V_1 \otimes V_2 = V)$$

(Dynamic)

$$[\underline{\mathit{lift}}]\ \frac{E \Downarrow \ell}{\$_{\mathsf{b}}E \Downarrow \$_{\mathsf{b}}\ell} \qquad [\underline{\mathit{var}}]\ \frac{}{x \Downarrow x} \qquad [\underline{\mathit{cst}}]\ \frac{}{\underline{d} \Downarrow \underline{d}} \qquad [\underline{\mathit{app}}]\ \frac{E_1 \Downarrow \mathcal{O}_1 \quad E_2 \Downarrow \mathcal{O}_2}{E_1 \underline{@} E_2 \Downarrow \mathcal{O}_1 \underline{@} \mathcal{O}_2}$$

$$[\underline{\mathit{lam}}]\ \frac{\Delta, y : \sigma, B \rhd [\cdot]E\{y/x\} \Downarrow [B']\mathcal{O} \quad y \notin \mathit{dom}\, B \cup \mathit{dom}\, \Delta}{\Delta \rhd [B]\underline{\lambda} x.E \Downarrow [B]\underline{\lambda} y.\underline{\mathbf{let}}^*\ B'\ \underline{\mathbf{in}}\ \mathcal{O}}$$

$$[\underline{\mathit{let}}]\ \frac{\begin{array}{c}\Delta \rhd [B]E_1 \Downarrow [B']\mathcal{O}_1 \quad \Delta, y : \sigma, B \rhd [\cdot]E_2\{y/x\} \Downarrow [B'']\mathcal{O}_2 \\ y \notin \mathit{dom}\, B' \cup \mathit{dom}\, \Delta\end{array}}{\Delta \rhd [B]\underline{\mathbf{let}}\ y \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 \Downarrow [B']\underline{\mathbf{let}}\ x \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ (\underline{\mathbf{let}}^*\ B''\ \underline{\mathbf{in}}\ \mathcal{O}_2)}$$

$$[\#]\ \frac{\Delta \rhd [B]E \Downarrow [B']\mathcal{O} \quad x \notin \mathit{dom}\, B' \cup \mathit{dom}\, \Delta}{\Delta \rhd [B]\#E \Downarrow [B', x : \sigma = \mathcal{O}]x}$$

Figure 9: The evaluation semantics of vPCF^{2}

Type Preservation: During the evaluation, the generated bindings B hold context information of the term E. The type preservation, therefore, uses a notion of typable binder-term-in-context, which extends the notion of typable term-in-context. A similar notion to binder-term-in-context has been used by

Hatcliff and Danvy to formalize continuation-based partial evaluation [20].

Definition 4.1 (Binder-term-in-context). *For a binder* $B \equiv (x_1 : \sigma_1 = \mathcal{O}_1, \ \ldots, x_n : \sigma_n = \mathcal{O}_n)$*, we write* $\Gamma \triangleright [B]E : \tau$ *if* $\Gamma, x_1 : ⓥ\sigma_1, \ldots, x_{i-1} : ⓥ\sigma_{i-1} \triangleright \mathcal{O}_i : ⓔ\sigma_i$ *for all* $1 \le i \le n$*, and* $\Gamma, x_1 : ⓥ\sigma_1, \ldots, x_n : ⓥ\sigma_n \triangleright E : \tau$*.*

Theorem 4.2 (Type preservation). *If* $ⓥ\Delta \triangleright [B]E : \tau$ *and* $\Delta \triangleright [B]E \Downarrow [B']V$*, then* $ⓥ\Delta \triangleright [B']V : \tau$*.*

The evaluation of a complete program inserts the bindings accumulated at the top level.

Definition 4.3 (Observation of complete program). *For a complete program* $\triangleright E : ⓔ\sigma$*, we write* $E \searrow \underline{\textbf{let}}^* \ B \ \underline{\textbf{in}} \ \mathcal{O}$ *if* $\triangleright [\cdot]E \Downarrow [B]\mathcal{O}$*.*

Corollary 4.4 (Type preservation for complete programs). *If* $\triangleright E : ⓔ\sigma$ *and* $E \searrow \mathcal{O}$*, then* $\triangleright \mathcal{O} : ⓔ\sigma$*.*

Semantic Correctness: The definition of erasure is straightforward and similar to the CBN case, and is thus omitted; the only important extra case is the erasure of trivialization: $|\#E| = |E|$.

Lemma 4.5 (Annotation erasure). *If* $\mathsf{vPCF}^2 \vdash ⓥ\Delta \triangleright [B]E : \tau$ *and* $\mathsf{vPCF}^2 \vdash \Delta \triangleright [B]E \Downarrow [B']V$*, then* $\mathsf{vPCF} \vdash \Delta \triangleright \textbf{let}^* \ |B| \ \textbf{in} \ |E| = \textbf{let}^* \ |B'| \ \textbf{in} \ |V| : |\tau|$*.*

Theorem 4.6 (Annotation erasure for complete programs). *If* $\mathsf{vPCF}^2 \vdash \triangleright E : ⓔ\sigma$ *and* $\mathsf{vPCF}^2 \vdash E \searrow \mathcal{O}$*, then* $\mathsf{vPCF} \vdash \ \triangleright |E| = |\mathcal{O}| : \sigma$*.*

Native embedding: Without going into detail, we remark that vPCF^2 has a simple native embedding $\{\!|-|\!\}_{\mathsf{v}\epsilon}$ into $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$, a CBV language with a term type and a state that consists of two references cells: We use one to hold the bindings and the other to hold a counter for generating fresh variables. As such, the language $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$ is a subset of ML; the language vPCF^2 can thus be embedded into ML, with dynamic constructs defined as functions.[9] The correctness proof for the embedding is by directly relating the derivation of the evaluation from a term E, in vPCF^2, and the derivation of the evaluation from its translation $\{\!|E|\!\}_{\mathsf{v}\epsilon}$, in $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$. The details of the native embedding and the accompanying correctness proof, again, are available in Appendix C.

4.3 Example: call-by-value type-directed partial evaluation

The problem specification of CBV TDPE is similar to the CBN TDPE, where the semantics is given by a translation into vPCF instead of nPCF. We only need to slightly modify the original formulation by inserting the trivialization operators $\#$ at appropriate places, so that the two-level program $NF(E)$ type

[9]The ML source code, with the following example of CBV TDPE, is available at the URL `www.brics.dk/~zheyang/programs/vPCF2`.

checks in vPCF^2. The call-by-value TDPE algorithm thus formulated is shown in Figure 10. We establish its semantic correctness, with respect to vPCF-equality this time, using a simple annotation erasure argument again; the proof is very similar to that of Theorem 2.12. Composing with the native embedding $\{\!|-|\!\}_{\mathsf{v}\epsilon}$, we have an efficient implementation of this formulation—which is essentially the call-by-value TDPE algorithm that uses state-based let-insertion [49]; see also Filinski's formal treatment [13].

a. The language of CBV TDPE: $\mathsf{vPCF}^{\mathsf{tdpe}}$

The syntax is the same as that of CBN TDPE, with the addition of a let-construct.

$$[\mathit{let}]\ \frac{\Delta, x{:}\,\sigma_1 \rhd E_2 : \sigma_2 \quad \Delta \rhd E_1 : \sigma_1}{\Delta \rhd \mathbf{let}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 : \sigma_2}$$

b. Standard instantiation (TDPE-erasure)

$$\boxed{\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi} \Longrightarrow \boxed{\mathsf{vPCF} \vdash |\Gamma| \rhd |E| : |\varphi|}$$

$$|\mathsf{b}^{\mathfrak{d}}| = \mathsf{b}; \qquad |\$_{\mathsf{b}} E| = |E|, |d^{\mathfrak{d}}| = d$$

c. Extraction functions $\downarrow^{\sigma}$ and $\uparrow_{\sigma}$:

We write $\sigma^{ⓥ}$ for the type $\sigma\{ⓥ\mathsf{b}/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$.

$$\begin{cases} \mathsf{vPCF}^2 \vdash \rhd{\downarrow^{\sigma}} : \sigma^{ⓥ} \to ⓥ\sigma \\ \downarrow^{\mathsf{b}} = \lambda x.x \\ \downarrow^{\sigma_1 \to \sigma_2} = \lambda f.\underline{\lambda} x.\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x)) \end{cases}$$

$$\begin{cases} \mathsf{vPCF}^2 \vdash \rhd{\uparrow_{\sigma}} : ⓥ\sigma \to \sigma^{ⓥ} \\ \uparrow_{\mathsf{b}} = \lambda x.x \\ \uparrow_{\sigma_1 \to \sigma_2} = \lambda e.\lambda x.\uparrow_{\sigma_2} \#(e\,\underline{@}(\downarrow^{\sigma_1} x)) \end{cases}$$

d. Residualizing instantiation

$$\boxed{\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi} \Rightarrow \boxed{\mathsf{vPCF}^2 \vdash \{\!|\Gamma|\!\}_{\mathsf{ri}} \rhd \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}_{\mathsf{ri}}}$$

$$\{\!|\mathsf{b}^{\mathfrak{d}}|\!\}_{\mathsf{ri}} = ⓥ\mathsf{b}; \qquad \{\!|\$_{\mathsf{b}} E|\!\}_{\mathsf{ri}} = \$_{\mathsf{b}}\{\!|E|\!\}_{\mathsf{ri}}, \{\!|d^{\mathfrak{d}} : \sigma^{\mathfrak{d}}|\!\}_{\mathsf{ri}} = \uparrow_{\sigma} d^{\mathfrak{d}}$$

e. The static normalization function

$$NF(\rhd E : \sigma^{\mathfrak{d}}) = \downarrow^{\sigma}\{\!|E|\!\}_{\mathsf{ri}} : ⓔ\sigma$$

Figure 10: Call-by-value type-directed partial evaluation

Syntactic correctness: The let-insertions slightly complicate the reasoning about which terms can be generated, since the point where the operator # is used does not lexically relate to the insertion point, where a residual binder is introduced. The refinement of the type system thus should also cover the types of the binders.

Figure 11 shows the refined type system; it is easy to prove that the code-typed values correspond to the object-level terms typable with the rules in Figure 12, which specify the λ_c-normal forms [13]. A term in λ_c-normal form can be either a normal value (*nv*) or a normal computation (*nc*). The other two syntactic categories that we use are atomic values (*av*; i.e., variables, literals, constants) and binders (*bd*, which must be an application of an atomic value to a normal value). Intuitively, evaluating terms in the refined type system can only introduce binding expressions whose types are of the form $ⓔ^{bd}(\sigma)$.

Definition 4.7 (Refined binder-term-in-context). *For a binder $B \equiv (x_1 : \sigma_1 = \mathcal{O}_1, \cdots, x_n : \sigma_n = \mathcal{O}_n)$, we write $\Gamma \blacktriangleright [B]E : \tau$ if $\Gamma, x_1 : ⓥ^{var}(\sigma_1), \ldots, x_{i-1} : ⓥ^{var}(\sigma_{i-1}) \blacktriangleright \mathcal{O}_i : ⓔ^{bd}(\sigma_i)$ for all $1 \leq i \leq n$, and $\Gamma, x_1 : ⓥ^{var}(\sigma_1), \ldots, x_n : ⓥ^{var}(\sigma_n) \blacktriangleright E : \tau$.*

Theorem 4.8 (Refined type preservation). *If $\mathsf{vPCF}^2 \vdash ⓥ^{var}(\Delta) \blacktriangleright [B]E : \tau$ and $\mathsf{vPCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$, then $\mathsf{vPCF}^2 \vdash ⓥ^{var}(\Delta) \blacktriangleright [B']V : \tau$.*

Corollary 4.9 (Refined type preservation for complete programs). *If $\blacktriangleright E : ⓔ^{nc}(\sigma)$ and $E \searrow \mathcal{O}$, then $\blacktriangleright \mathcal{O} : ⓔ^{nc}(\sigma)$.*

Theorem 4.10 (Normal-form code types). *If V is an vPCF^2-value (Figure 8), and $\mathsf{vPCF}^2 \vdash ⓥ^{var}(\Delta) \blacktriangleright V : ⓥ^X(\sigma)$ where X is av, nv, bd, or nc, then $V \equiv \mathcal{O}$ for some $\mathcal{O}$ and $\Delta \rhd^X |\mathcal{O}| : \sigma$.*

To show that the CBV TDPE algorithm only generates term in λ_c-normal form, it suffices to show its typability with respect to the refined type system.

Lemma 4.11. *(1) The extraction functions (Figure 10c) have the following normal-form types (writing $\sigma^{\bigcirc\mathsf{nv}}$ for $\sigma\{ⓥ^{nv}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$.)*

$$\blacktriangleright \downarrow^{\sigma} : \sigma^{\bigcirc\mathsf{nv}} \to ⓥ^{nv}(\sigma), \blacktriangleright \uparrow_{\sigma} : ⓥ^{av}(\sigma) \to \sigma^{\bigcirc\mathsf{nv}}.$$

(2) If $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi$, then $\mathsf{vPCF}^2 \vdash \{\!|\Gamma|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}^{\mathsf{nv}}_{\mathsf{ri}}$, where $\{\!|\varphi|\!\}^{\mathsf{nv}}_{\mathsf{ri}} = \varphi\{ⓥ^{nv}(\mathsf{b})/\mathsf{b}^{\mathfrak{d}} : \mathsf{b} \in \mathbb{B}\}$.

Theorem 4.12. *If $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, then $\mathsf{vPCF}^2 \vdash\blacktriangleright NF(E) : ⓥ^{nv}(\sigma)$.*

5 Related work

The introduction (Section 1) of this article has already touched upon some related work, which forms the general background of this work. Here we examine other related work in the rich literature of two-level languages, and put the current work in perspective.

Types $\tau \ ::= \ \theta \mid ⓔ^{bd}(\sigma) \mid ⓔ^{nc}(\sigma)$

$\theta \ ::= \ \mathsf{b} \mid ⓥ^{var}(\sigma) \mid ⓥ^{nv}(\sigma) \mid ⓥ^{av}(\sigma) \mid \theta \to \tau$

$\sigma \ ::= \ \mathsf{b} \mid \sigma_1 \to \sigma_2$

Typing Judgment $\boxed{\mathsf{vPCF}^2 \vdash \Gamma \blacktriangleright E : \tau}$

(Static) same as the static rules for $\mathsf{vPCF}^2 \vdash \Gamma \rhd E : \tau$

(Dynamic)

$$\frac{\Gamma \blacktriangleright E : ⓥ^{var}(\sigma)}{\Gamma \blacktriangleright E : ⓥ^{av}(\sigma)} \qquad \frac{Sg(d) = \sigma}{\Gamma \blacktriangleright \underline{d} : ⓥ^{av}(\sigma)} \qquad \frac{\Gamma \blacktriangleright E : \mathsf{b}}{\Gamma \blacktriangleright \$_{\mathsf{b}} E : ⓥ^{av}(\mathsf{b})}$$

$$\frac{\Gamma \blacktriangleright E : ⓥ^{av}(\mathsf{b})}{\Gamma \blacktriangleright E : ⓥ^{nv}(\mathsf{b})} \qquad \frac{\Gamma, x : ⓥ^{var}(\sigma_1) \blacktriangleright E : ⓔ^{nc}(\sigma_2)}{\Gamma \blacktriangleright \underline{\lambda} x.E : ⓥ^{nv}(\sigma_1 \to \sigma_2)}$$

$$\frac{\Gamma \blacktriangleright E_1 : ⓥ^{av}(\sigma_2 \to \sigma) \quad \Gamma \blacktriangleright E_2 : ⓥ^{nv}(\sigma_2)}{\Gamma \blacktriangleright E_1 \underline{@} E_2 : ⓔ^{bd}(\sigma)} \qquad \frac{\Gamma \blacktriangleright E : ⓥ^{nv}(\sigma)}{\Gamma \blacktriangleright E : ⓔ^{nc}(\sigma)}$$

$$\frac{\Gamma, x{:}ⓥ^{var}(\sigma_1) \blacktriangleright E_2 : ⓔ^{nc}(\sigma_2) \quad \Gamma \blacktriangleright E_1 : ⓔ^{bd}(\sigma_1)}{\Gamma \blacktriangleright \underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 : ⓔ^{nc}(\sigma_2)}$$

$$\frac{\Gamma \blacktriangleright E : ⓔ^{bd}(\sigma)}{\Gamma \blacktriangleright \# E : ⓥ^{var}(\sigma)}$$

Figure 11: vPCF²-terms that generate code in λ_c-normal form

$$\frac{x{:}\sigma \in \Delta}{\Delta \rhd^{av} x : \sigma} \qquad \frac{Sg(d) = \sigma}{\Delta \rhd^{av} d : \sigma} \qquad \frac{\ell \in \mathbb{L}(\mathsf{b})}{\Delta \rhd^{av} \ell : \mathsf{b}} \qquad \frac{\Delta \rhd^{av} E : \mathsf{b}}{\Delta \rhd^{nv} E : \mathsf{b}}$$

$$\frac{\Delta, x : \sigma_1 \rhd^{nc} E : \sigma_2}{\Delta \rhd^{nv} \lambda x.E : \sigma_1 \to \sigma_2} \qquad \frac{\Delta \rhd^{av} E_1 : \sigma_2 \to \sigma \quad \Delta \rhd^{nv} E_2 : \sigma_2}{\Delta \rhd^{bd} E_1\, E_2 : \sigma}$$

$$\frac{\Delta \rhd^{nv} E : \sigma}{\Delta \rhd^{nc} E : \sigma} \qquad \frac{\Delta, x : \sigma_1 \rhd^{nc} E_2 : \sigma_2 \quad \Delta \rhd^{bd} E_1 : \sigma_1}{\Delta \rhd^{nc} \mathbf{let}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 : \sigma_2}$$

Figure 12: Inference rules for terms in λ_c-normal form

5.1 Two-level formalisms for compiler construction

While Jones et al. [23, 24, 25] studied two-level languages mainly as meta-languages for expressing partial evaluators and proving them correct, Nielson and Nielson's work explored various other aspects and applications of two-level languages, such as the following ones.

- A formalism for components of compiler backend, in particular code generation and abstract interpretation, and associated analysis algorithms [40]. These two-level languages embrace a traditional view of code objects—as closed program fragments of function type; name capturing is therefore not an issue in such a setting. By design, these two-level languages are intended as meta-languages for combinator-based code generators, as have been used, e.g., by Wand [51]. In contrast, in meta-languages for partial evaluators and higher-order code generators (such as the examples studied in the present article), it is essential to be able to manipulate open code, i.e., code with free variables: Without this ability, basic transformations such as unfolding (a.k.a. inlining) would rarely be applicable.

- A general framework for the type systems of two-level and multi-level languages, which, on the descriptive side [41], provides a setting for comparing and relating such languages, and, on the prescriptive side [42], offers guidelines for designing new such languages. Their investigation, however, stopped short at the type systems, which are not related to any semantics. Equipping their framework of two-level types systems with some general form of semantics in the spirit of this article, if possible, seems a promising step towards practicality.

To accommodate such a wide range of applications, Nielson and Nielson developed two-level languages syntactically and used parameterized semantics. In contrast, the framework in the present article generalizes the two-level languages of Jones et al., where specific semantic properties such as annotation erasure are essential to the applications. These two lines of studies complement each other.

Beyond the study of two-level languages, two-level formalisms abound in the literature of semantics-based compiler construction. Morris showed how to refine Landin's semantics [29], viewed as an interpreter, into a compiler [38]. Mosses developed action semantics [39] as an alternative to denotational semantics. An action semantics defines a compositional translation of programs into actions, which are primitives whose semantics can be concisely defined. The translation can be roughly viewed as a two-level program, the dynamic part of which is composed of actions. Lee successfully demonstrated how this idea can be used to construct realistic compilers [31].

5.2 Correctness of partial evaluators

As mentioned in the introduction, annotation erasure has long been used to formalize the correctness of partial evaluation, but existing work on proving annotation erasure while modeling the actual, name-generation-based implementation, used denotational semantics and stayed clear of operational semantics. Along this line, Gomard used a domain-theoretical logical relation to prove annotation erasure [16], but he treated fresh name generation informally. Moggi gave a formal proof, using a functor category semantics to model name generation [36]. Filinski established a similar result as a corollary of the correctness of type-directed partial evaluation, the proof of which, in turn, used an ω-admissible Kripke logical relation in a domain-theoretical semantics [12, Section 5.1]. The present work, in contrast, factors the realistic name-generation-based implementations through native embeddings from high-level, substitution-based operational semantics. In the high-level operational semantics, simple elementary reasoning often suffices for establishing semantics properties such as annotation erasure, as demonstrated in this article.

Wand proved the correctness of Mogensen's compact partial evaluator for pure λ-calculus using a logical relation that, at the base type, amounts to an equational formulation of annotation erasure [52, Theorem 2]. Mogensen's partial evaluator encodes two-level terms as λ-terms, employing higher-order abstract syntax for representing bindings. In this λ-calculus-based formulation, the generation of residual code is modeled using symbolic normalization in the λ-calculus.

Palsberg [43] presented another correctness result for partial evaluation, using a reduction semantics for the two-level λ-calculus. Briefly, his result states that static reduction does not go wrong and generates a static normal form. In the pure λ-calculus, where reductions are confluent, this correctness result implies annotation erasure.

5.3 Macros and syntactic abstractions

The code-type view of two-level languages, in its most rudimentary form, can be traced back to the S-expressions of Lisp [21]. Since S-expressions serve as a representation for both programs and data, they popularized Lisp as an ideal test bed for experimenting program analysis and synthesis. One step further, the quasiquote/unquote mechanism [1] offers a succinct and intuitive notation for code synthesis, one that makes the staging information explicit.

The ability of expressing program manipulation concisely then led to introducing the mechanism of macros in Lisp, which can be informally understood as the compile-time execution of two-level programs. Practice, soon, revealed the problem of name-capturing in the generated code. A proper solution of this problem, namely hygienic macro expansion [27, 28], gained popularity in various Scheme dialects. Having been widely used to build language extensions of Scheme, and even domain-specific languages on top of Scheme, hygienic macros have evolved into syntactic abstractions, now part of the Scheme standard [26].

The studies of two-level languages could pave the way to a future generation of macro languages. The most prominent issue of using macros in Scheme is the problem of debugging. It divides into debugging the syntax of the macro-expanded program (to make it well-formed) and debugging the semantics of macro-expanded programs (to ensure that it runs correctly). These two tasks are complicated by the non-intuitive control flow introduced by the staging. In the light of two-level languages, these two tasks correspond to the syntactic and semantic correctness of generated code. Therefore, if we use two-level languages equipped with the properties studied in this article, then we can address these two tasks:

- for the syntax of macro-expanded programs, type checking in the two-level language provides static debugging; and

- for the semantics of macro-expanded programs, we can reduce debugging the macro (a two-level function) to debugging a normal function—its erasure.

To make two-level languages useful as syntactic-abstraction languages in the style of Scheme, the key extensions seem to be multiple syntactic categories and suitable concrete syntax.

5.4 Multi-level languages

Many possible extensions and variations of two-level languages exist. Going beyond the two-level stratification, we have the natural generalization of multi-level languages. While this generalization, by itself, accommodates few extra practical applications,[10] its combination with a *run* construct holds a greater potential. The run construct allows immediate execution of the generated code; therefore, code generation and code execution could happen during a single evaluation phase—this ability, often called run-time code generation, has a growing number of applications in system areas [53].

Davies and Pfenning investigated multi-level languages through the Curry-Howard correspondence with modal logics: $\lambda^{\Box}$, which corresponds to intuitionistic modal logic S4, has the run construct, but it can only manipulates closed code fragment [11]; $\lambda^{\bigcirc}$, which corresponds to linear temporal logic, can manipulate open code fragment, but does not have the run construct [10]. Naively combining the two languages would result in an unsound type system, due to the execution of program fragments with unbound variables. Moggi et al.'s Idealized MetaML [37] combines $\lambda^{\Box}$ and $\lambda^{\bigcirc}$, by ensuring that the argument to the run-construct be properly closed. Calcagno et al. further studied how side effects can be added to Idealized MetaML while retaining type soundness [4].

While the development of various multi-level languages has been centered on the conflicts of expressiveness and type soundness, other important aspects of multi-level languages, such as efficient code generation and formal reasoning, have not been much explored. Wickline et al. formalized an efficient implementation of $\lambda^{\Box}$ in terms of an abstract machine [53]. Taha axiomatized a fragment of Idealized MetaML, which can be used for equational reasoning [50].

For a practical multi-level language, both efficient code generation and formal support of reasoning and debugging would be crucial. It is interesting to see whether the work in this article can be extended to multi-level languages similar to Idealized MetaML in expressiveness, yet equipped with an efficient implementation for code generation, and the erasure property (probably for restricted fragments of the languages).

[10] It would be, however, interesting to see whether real-life applications like the automake suite in Unix can be described as three-level programs.

5.5 Applications

Two-level languages originate as a formalism of partial evaluation, while erasure property captures the correctness of partial evaluation. Consequently, many standard applications of partial evaluation can be modeled as two-level programs: For example, automatic compilation by specializing an interpreter (which is known as the first Futamura projection [15]) can be achieved with a two-level program—the staged interpreter. The erasure property reduces the correctness of automatic compilation to that of the interpreter.

Some applications are explained and analyzed using the technique of partial evaluation, but not realized through a dedicated partial evaluator. The one-pass CPS transformer of Danvy and Filinski is one such example. In this case, it is not a whole program, but the output of a transformation (Plotkin's CPS translformation), to be binding-time analyzed. The explicit staging of two-level languages makes them the ideal candidate for describing such algorithms. The technique of Section 2.2, for example, can be used for constructing other one-pass CPS transformations and proving them correct: e.g., call-by-name CPS transformation (see Appendix A) and CPS transformation of programs after strictness analysis [8]. We have also applied this technique to stage other monadic transformations (such as a state-passing-style transformation) into one-pass versions that avoid the generation of administrative redexes.

The two-level language vPCF^2 (and similarly, nPCF^2) can also be used to account for the self application of partial evaluators. Under the embedding translation, a vPCF^2-program becomes a one-level program in $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$, which is a language with computational effects, and an instance of the object language vPCF of vPCF^2. With some care, it is not difficult to derive a self application based on this idea and prove it correct. In fact, such a process has been developed in detail (though without using the two-level language formalism of this article) for self-applying TDPE to produce efficient generating extensions [17], which is known as the second Futamura projection.

In recent years, type systems have been used to capture, in a syntactic fashion, a wide range of language notions, such as security and mobility. It seems possible to apply the code-type-refinement technique (Sections 2.6 and 4.3) to guarantee that code generated by a certain (possibly third-party) program is typable in such a type system; this could lead to the addition of a code-generating dimension to the area of trusted computing.

6 Conclusions

6.1 Summary of contributions

We have pinpointed several properties of two-level languages that are useful for reasoning about semantic and syntactic properties of code generated by two-level programs, and for providing them with efficient implementations. More specifically, we have made the following technical contributions.

- We have proved annotation erasure for both languages, using elementary equational reasoning, and the proofs are simpler than those in previous works, which use denotational formulations and logical relations *directly* (i.e., which do not factor out a native embedding from a two-level language). On the technical side, our proofs take advantage of the fact that the substitution operations used in the operational semantics of the two-level languages do not capture dynamic bound variables.

- We have constructed native embeddings of both languages into conventional languages and proved them correct, thereby equipping the two-level semantics with efficient, substitution-free implementations. To our knowledge, such a formal connection between the symbolic semantics and its implementation has not been established for other two-level languages [10, 37, 40].

- We have formulated the one-pass call-by-value CPS transformation, call-by-name TDPE, and call-by-value TDPE in these two-level languages. Through the native embeddings, they match Danvy and Filinski's original work. We also have formulated other one-pass CPS transformations and one-pass transformations into monadic style, for given monads. We use annotation erasure to prove the semantic correctness of these algorithms.

  To our knowledge, the present paper is the first to formally use annotation erasure to prove properties of hand-written programs—as opposed to two-level programs used internally by partial evaluation. Previously, annotation erasure has been informally used to motivate and reason about such programs.

  The formulation of TDPE as translations from the special two-level languages for TDPE to conventional two-level languages also clarifies the relationship between TDPE and traditional two-level formulations of partial evaluation, which was an open problem. In practice, this formal connection implies that it is sound to use TDPE in a conventional two-level framework for partial evaluation, e.g., to perform higher-order lifting—one of the original motivations of TDPE [5].

- We have proved the syntactic correctness of both call-by-name TDPE and call-by-value TDPE—i.e., that they generate terms in long $\beta\eta$-normal form and terms in λ_c-normal form, respectively—by showing type preservation for refined type systems where code-typed values are such terms, and that the corresponding TDPE algorithms are typable in the refined type systems.

  The semantic and syntactic correctness results about TDPE match Filinski's results [12, 13], which have been proved from scratch using denotational methods.

6.2 Direction for future work

It would be interesting to see whether and how far our general framework (Section 3) can apply to other scenarios, e.g., where the object language is a concurrent calculus, equationally specified. As we have seen in Section 5, it seems promising to combine our framework with the related work, and to find applications in it.

For the specific two-level languages developed in this article, immediate future work could include:

- to establish a general theorem for refined type preservation;
- to find and prove other general properties: For example, an adequacy theorem for two-level evaluation with respect to the one-level equational theory could complete our account of TDPE with a completeness result, which says that if there is a normal form, TDPE will terminate and find it; and,
- to further explore the design space of two-level languages by adding an online dimension to them (in the sense of "online partial evaluation"): For example, we could consider adding interpreted object-level constants to the two-level language, which are expressed through equations in the type theory of the one-level language. The extra information makes it possible to generate code of a higher quality.

A Call-by-name CPS translation

Danvy and Filinski [7] also presented a one-pass version for Plotkin's call-by-name CPS transformation. Figure 13 shows both transformations. The erasure argument applies here, too.

Other one-pass CPS transformations [19] can be similarly described and proven correct.

a. Source syntax: the pure simply typed λ-calculus $\mathsf{n\Lambda}$

Types $\sigma ::= \mathsf{b} \mid \sigma_1 \to \sigma_2$

Raw terms $E ::= x \mid \lambda x.E \mid E_1\, E_2$

Typing judgment $\boxed{\mathsf{n\Lambda} \vdash \Delta \rhd E : \sigma}$ (omitted)

b. Plotkin's original transformation:

$$\boxed{\mathsf{n\Lambda} \vdash \Delta \rhd E : \sigma} \Longrightarrow \boxed{\mathsf{nPCF} \vdash K\{\!\!\{\Delta\}\!\!\}_{\mathsf{p}\kappa} \rhd \{\!\!\{E\}\!\!\}_{\mathsf{p}\kappa} : K\{\!\!\{\sigma\}\!\!\}_{\mathsf{p}\kappa}}.$$

Here, $K\sigma = (\sigma \to \mathsf{Ans}) \to \mathsf{Ans}$ for an answer type Ans.

Types: $\{\!\!\{\mathsf{b}\}\!\!\}_{\mathsf{p}\kappa} = \mathsf{b}$,

$\{\!\!\{\sigma_1 \to \sigma_2\}\!\!\}_{\mathsf{p}\kappa} = K\{\!\!\{\sigma_1\}\!\!\}_{\mathsf{p}\kappa} \to K\{\!\!\{\sigma_2\}\!\!\}_{\mathsf{p}\kappa}$

Terms: $\{\!\!\{x\}\!\!\}_{\mathsf{p}\kappa} = \lambda k.x\, k$,

$\{\!\!\{\lambda x.E\}\!\!\}_{\mathsf{p}\kappa} = \lambda k.k\, \lambda x.\{\!\!\{E\}\!\!\}_{\mathsf{p}\kappa}$

$\{\!\!\{E_1\, E_2\}\!\!\}_{\mathsf{p}\kappa} = \lambda k.\{\!\!\{E_1\}\!\!\}_{\mathsf{p}\kappa} \lambda r_1.r_1\, \{\!\!\{E_2\}\!\!\}_{\mathsf{p}\kappa}\, k.$

c. Danvy and Filinski's one-pass transformation:

The transformation is specified as a pair of mutually recursive translations.

1. The (higher-order) auxiliary translation

$$\boxed{\mathsf{n\Lambda} \vdash \Delta \rhd E : \sigma} \Longrightarrow \boxed{\mathsf{nPCF^2} \vdash \bigcirc(K\{\!\!\{\Delta\}\!\!\}_{\mathsf{p}\kappa}) \rhd \{\!\!\{E\}\!\!\}_{\mathsf{df^2}\kappa} : K^{\bigcirc}(\bigcirc\{\!\!\{\sigma\}\!\!\}_{\mathsf{p}\kappa})}$$

Here, $K^{\bigcirc}\sigma = (\sigma \to \bigcirc\mathsf{Ans}) \to \bigcirc\mathsf{Ans}$.

$$\begin{cases} \{\!\!\{x\}\!\!\}_{\mathsf{df^2}\kappa} & = & \lambda k.x \,\underline{@}\, \underline{\lambda} y.k\, y \\ \{\!\!\{\lambda x.E\}\!\!\}_{\mathsf{df^2}\kappa} & = & \lambda k.k\, \underline{\lambda} x.\{\!\!\{E\}\!\!\}_{\mathsf{df}\kappa} \\ \{\!\!\{E_1\, E_2\}\!\!\}_{\mathsf{df^2}\kappa} & = & \lambda k.\{\!\!\{E_1\}\!\!\}_{\mathsf{df^2}\kappa}\, \lambda r_1.r_1 \underline{@} \{\!\!\{E_2\}\!\!\}_{\mathsf{df}\kappa} \underline{@}\, \underline{\lambda} x.k\, x \end{cases}$$

2. The complete translation

$$\Longrightarrow \boxed{\mathsf{nPCF^2} \vdash \bigcirc(K\{\!\!\{\Delta\}\!\!\}_{\mathsf{p}\kappa}) \rhd \{\!\!\{E\}\!\!\}_{\mathsf{df}\kappa} : \bigcirc(K\{\!\!\{\sigma\}\!\!\}_{\mathsf{p}\kappa})}$$

$$\begin{cases} \{\!\!\{x\}\!\!\}_{\mathsf{df}\kappa} & = & x \\ \{\!\!\{\lambda x.E\}\!\!\}_{\mathsf{df}\kappa} & = & \underline{\lambda} k.k \underline{@}\, \underline{\lambda} x.\{\!\!\{E\}\!\!\}_{\mathsf{df}\kappa} \\ \{\!\!\{E_1\, E_2\}\!\!\}_{\mathsf{df}\kappa} & = & \underline{\lambda} k.\{\!\!\{E_1\}\!\!\}_{\mathsf{df^2}\kappa}\, \lambda r_1.r_1 \underline{@} \{\!\!\{E_2\}\!\!\}_{\mathsf{df}\kappa} \underline{@} k \end{cases}$$

Figure 13: Call-by-name CPS transformation

B Expanded proofs for nPCF^2

B.1 Type preservation and annotation erasure

Theorem 2.1 (Type preservation). *If* $\bigcirc\Delta \rhd E : \tau$ *and* $E \Downarrow V$*, then* $\bigcirc\Delta \rhd V : \tau$.

Proof. Induction on $E \Downarrow V$. For the only non-straightforward case, where $E \equiv E_1\, E_2$, we use a Substitution Lemma of the typing rules: if $\Gamma, x : \tau_1 \rhd E : \tau_2$ and $\Gamma \rhd E' : \tau_1$, then $\Gamma \rhd E\{E'/x\} : \tau_2$. □

Theorem 2.3 (Annotation erasure). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF} \vdash \Delta \rhd |E| = |V| : |\tau|$.

Its proof uses the following Substitution Lemma for erasure.

Lemma B.1 (Substitution lemma for $|-|$). *If* $\mathsf{nPCF}^2 \vdash \Gamma, x : \tau' \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash \Gamma \rhd E' : \tau'$*, then* $|E\{E'/x\}| \sim_\alpha |E|\{|E'|/x\}$.

Proof. By a simple induction on the size of term E. □

Proof of Theorem 2.3. By rule induction on $E \Downarrow V$. We show a few cases.

Case [*app*]:
$$\begin{aligned} & |E_1\, E_2| \equiv |E_1|\, |E_2| \\ \stackrel{i.h.}{=}\ & (\lambda x.|E'|)\, |E_2| = |E'|\{|E_2|/x\} \\ \sim_\alpha\ & |E'\{E_2/x\}| && (\textit{Lemma B.1}) \\ \stackrel{i.h.}{=}\ & |V|. \end{aligned}$$

Case [*fix*]: $|\mathbf{fix}\, E| \equiv \mathbf{fix}\, |E| = |E|(\mathbf{fix}\, |E|) \equiv |E(\mathbf{fix}\, E)| \stackrel{i.h.}{=} |V|$.

Case [$\underline{lam}$]: $|\underline{\lambda} x.E| \equiv \lambda x.|E| \stackrel{i.h.}{=} \lambda x.|\mathcal{O}| \equiv |\underline{\lambda} x.\mathcal{O}|$. □

B.2 Native embedding

We first present the standard denotational semantics of the language nPCF^Λ.

Definition B.2. *(Denotational semantics of nPCF^Λ) Let* $\mathbf{Z}$ *and* $\mathbf{B}$ *denote the sets (discrete cpos) of integers and of booleans, respectively. Let* $\mathbf{Cst}$ *denote the set used to represent constants. Let* $\mathbf{E}$ *be the inductive set given as the smallest solution to the equation* $X = \mathbf{Z} + \mathbf{Z} + \mathbf{B} + \mathbf{Cst} + \mathbf{Z} \times X + X \times X$*, with injection functions* inVar*,* $\mathit{inLit}_{\mathsf{int}}$*,* $\mathit{inLit}_{\mathsf{bool}}$*,* inCst*,* inLam*, and* inApp *into the components of the sum.*

The standard domain-theoretical semantics maps nPCF^Λ-types to domains as follows.

$$[\![\mathsf{int}]\!] = \mathbf{Z}_\perp, [\![\mathsf{bool}]\!] = \mathbf{B}_\perp, [\![\Lambda]\!] = \mathbf{E}_\perp, [\![\sigma_1 \to \sigma_2]\!] = [\![\sigma_1]\!] \to [\![\sigma_2]\!]$$

This mapping extends to provide the meaning of contexts Δ by taking the product: $[\![\Delta]\!] = \prod_{x \in dom\,\Delta} [\![\Delta(x)]\!]$. The meaning of a term-in-context $\Delta \rhd E : \sigma$ is a continuous function $[\![E]\!] : [\![\Delta]\!] \to [\![\sigma]\!]$:

$$
\begin{aligned}
[\![\ell]\!]\rho &= \mathbf{val}^{\perp}\, \ell \\
[\![x]\!]\rho &= \rho x \\
[\![\lambda x.E]\!]\rho &= \lambda y.[\![E]\!]\rho[x \mapsto y] \\
[\![E_1\, E_2]\!]\rho &= [\![E_1]\!]\rho\,([\![E_2]\!]\rho) \\
[\![\mathbf{fix}\, E]\!]\rho &= \textstyle\bigsqcup_{i \geq 0} ([\![E]\!]\rho)^i(\perp) \\
[\![\mathbf{if}\, E_1\, E_2\, E_3]\!]\rho &= \mathbf{let}^{\perp}\, b \Leftarrow [\![E_1]\!]\rho \,\mathbf{in}\, \mathbf{if}\, b\,([\![E_2]\!]\rho)\,([\![E_3]\!]\rho) \\
[\![E_1 \otimes E_2]\!]\rho &= \mathbf{let}^{\perp}\, m \Leftarrow [\![E_1]\!]\rho \,\mathbf{in}\, \mathbf{let}^{\perp}\, n \Leftarrow [\![E_2]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(m \otimes n) \\
[\![\mathsf{VAR}(E)]\!]\rho &= \mathbf{let}^{\perp}\, i \Leftarrow [\![E]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(\mathit{inVar}(i)) \\
[\![\mathsf{LIT}_{\mathsf{b}}(E)]\!]\rho &= \mathbf{let}^{\perp}\, l \Leftarrow [\![E]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(\mathit{inLit}_{\mathsf{b}}(l)) \\
[\![\mathsf{CST}(E)]\!]\rho &= \mathbf{let}^{\perp}\, c \Leftarrow [\![E]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(\mathit{inCst}(c)) \\
[\![\mathsf{LAM}(E_1, E_2)]\!]\rho &= \mathbf{let}^{\perp}\, x \Leftarrow [\![E_1]\!]\rho \,\mathbf{in}\, \mathbf{let}^{\perp}\, e \Leftarrow [\![E_2]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(\mathit{inLam}(x, e)) \\
[\![\mathsf{VAR}(E_1, E_2)]\!]\rho &= \mathbf{let}^{\perp}\, e_1 \Leftarrow [\![E_1]\!]\rho \,\mathbf{in}\, \mathbf{let}^{\perp}\, e_2 \Leftarrow [\![E_2]\!]\rho \,\mathbf{in}\, \mathbf{val}^{\perp}\,(\mathit{inApp}(e_1, e_2))
\end{aligned}
$$

It is straightforward to show that the equational theory is sound with respect to this denotational semantics.

Theorem B.3 (Soundness of the equational theory). *If* $\mathsf{nPCF}^{\wedge} \vdash \Delta \rhd E_1 = E_2 : \sigma$*, then* $[\![E_1]\!] = [\![E_2]\!]$.

We now prove the correctness of the embedding translation, i.e., that evaluating complete programs of code type in nPCF^{2} is precisely simulated by evaluating their embedding translations in $\mathsf{nPCF}^{\wedge}$. We proceed in two steps. First, we show that if nPCF^{2}-evaluation of a term E generates certain object term as the result, then $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!]$ should give the encoding of this term, modulo α-conversion. Second, we show that conversely, if $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] \neq \perp$, then evaluation of term E terminates.

Lemma 2.6 (Substitution lemma for $\{\!|-|\!\}_{\mathsf{n}\epsilon}$). *If* $\mathsf{nPCF}^{2} \vdash \Gamma, x : \tau' \rhd E : \tau$ *and* $\mathsf{nPCF}^{2} \vdash \Gamma \rhd E' : \tau'$*, then* $\{\!|E\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \sim_\alpha \{\!|E|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\}$.

Proof. By induction on the size of term E. The most non-trivial case is the following one.

Case $E \equiv \underline{\lambda} y.E_1$: There are two sub-cases: Either $x \equiv y$ or $x \not\equiv y$. If $x \equiv y$, then

$$
\begin{aligned}
&\{\!|(\underline{\lambda} y.E_1)\{E'/y\}|\!\}_{\mathsf{n}\epsilon} \equiv \{\!|(\underline{\lambda} y.E_1)|\!\}_{\mathsf{n}\epsilon} \equiv \underline{\lambda}^{\mathsf{n}\epsilon}(\lambda y.\{\!|E_1|\!\}_{\mathsf{n}\epsilon}) \\
&\equiv (\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda y.\{\!|E_1|\!\}_{\mathsf{n}\epsilon}))\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/y\} \equiv \{\!|(\underline{\lambda} y.E_1)|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/y\}
\end{aligned}
$$

If $x \not\equiv y$, then let z be a variable such that $z \notin \mathit{fv}(E') \cup \{x\}$

$$
\begin{aligned}
&\{\!|(\underline{\lambda} y.E_1)\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \\
\sim_\alpha\ &\{\!|\underline{\lambda} z.E_1\{z/y\}\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \\
\equiv\ &\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda z.\{\!|E_1\{z/y\}\{E'/x\}|\!\}_{\mathsf{n}\epsilon}) \\
\sim_\alpha\ &\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda z.(\{\!|E_1\{z/y\}|\!\}_{\mathsf{n}\epsilon}\{E'/x\})) && (\textit{ind. hyp. on } E_1\{z/y\}) \\
\equiv\ &(\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda z.\{\!|E_1\{z/y\}|\!\}_{\mathsf{n}\epsilon}))\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\} && (z \notin \mathit{fv}(E') = \mathit{fv}(\{\!|E'|\!\}_{\mathsf{n}\epsilon})) \\
\sim_\alpha\ &\{\!|(\underline{\lambda} y.E_1)|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\}
\end{aligned}
$$

$\square$

Lemma 2.7 (Evaluation preserves translation). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \triangleright E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF}^\Lambda \vdash \{\!|\bigcirc\Delta|\!\}_{\mathsf{n}\epsilon} \triangleright \{\!|E|\!\}_{\mathsf{n}\epsilon} = \{\!|V|\!\}_{\mathsf{n}\epsilon} : \{\!|\tau|\!\}_{\mathsf{n}\epsilon}$.

Proof. By rule induction on $\mathsf{nPCF}^2 \vdash E \Downarrow V$. We show a few cases.

Case $[\mathit{app}]$**:**

$$\begin{array}{cll} & \{\!|E_1\,E_2|\!\}_{\mathsf{n}\epsilon} \equiv \{\!|E_1|\!\}_{\mathsf{n}\epsilon}\,\{\!|E_2|\!\}_{\mathsf{n}\epsilon} & \\ \stackrel{i.h.}{=} & \{\!|\lambda x.E'|\!\}_{\mathsf{n}\epsilon}\{\!|E_2|\!\}_{\mathsf{n}\epsilon} \equiv (\lambda x.\{\!|E'|\!\}_{\mathsf{n}\epsilon})\,\{\!|E_2|\!\}_{\mathsf{n}\epsilon} & \\ = & \{\!|E'|\!\}_{\mathsf{n}\epsilon}\{\{\!|E_2|\!\}_{\mathsf{n}\epsilon}/x\} & (\beta) \\ \sim_\alpha & \{\!|E'\{E_2/x\}|\!\}_{\mathsf{n}\epsilon} & (\textit{Lemma 2.6}) \\ \stackrel{i.h.}{=} & \{\!|V|\!\}_{\mathsf{n}\epsilon}. & \end{array}$$

Case $[\mathit{fix}]$**:**

$$\begin{array}{cll} & \{\!|\mathbf{fix}\,E|\!\}_{\mathsf{n}\epsilon} \equiv \mathbf{fix}\,\{\!|E|\!\}_{\mathsf{n}\epsilon} & \\ = & \{\!|E|\!\}_{\mathsf{n}\epsilon}\,(\mathbf{fix}\,\{\!|E|\!\}_{\mathsf{n}\epsilon}) & (\textit{equational rule for } \mathbf{fix}\,) \\ \equiv & \{\!|E\,(\mathbf{fix}\,E)|\!\}_{\mathsf{n}\epsilon} \stackrel{i.h.}{=} \{\!|V|\!\}_{\mathsf{n}\epsilon}. & \end{array}$$

The term-building evaluation of the dynamic parts preserves translation, because the translation of the dynamic constructs is compositional.

Case $[\underline{\mathit{lam}}]$**:** $\{\!|\underline{\lambda} x.E|\!\}_{\mathsf{n}\epsilon} \equiv \underline{\lambda}^{\mathsf{n}\epsilon}(\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon}) \stackrel{i.h.}{=} \underline{\lambda}^{\mathsf{n}\epsilon}(\lambda x.\{\!|\mathcal{O}|\!\}_{\mathsf{n}\epsilon}) \equiv \{\!|\underline{\lambda} x.\mathcal{O}|\!\}_{\mathsf{n}\epsilon}$. $\square$

Lemma 2.8 (Translation of code-typed value). *If* $\mathsf{nPCF}^2 \vdash v_1{:}\bigcirc\sigma_1, \ldots, v_n{:}\bigcirc\sigma_n \triangleright \mathcal{O} : \bigcirc\sigma$*, then there is a value* $t : \Lambda$ *such that*

- $\mathsf{nPCF}^\Lambda \vdash\ \triangleright (\{\!|\mathcal{O}|\!\}_{\mathsf{n}\epsilon}(n+1))\{\lambda i.\mathsf{VAR}(1)/v_1, \ldots, \lambda i.\mathsf{VAR}(n)/v_n\} = t : \Lambda,$
- $\mathsf{nPCF} \vdash v_1 : \sigma_1, \ldots, v_n : \sigma_n \triangleright \mathcal{D}(t) : \sigma$*, and*
- $|\mathcal{O}| \sim_\alpha \mathcal{D}(t)$.

Proof. By induction on the size of term $\mathcal{O}$. We write θ_n for the substitution $\{\lambda i.\mathsf{VAR}(1)/v_1, \ldots, \lambda i.\mathsf{VAR}(n)/v_n\}$.

Case $\mathcal{O} \equiv \$_{\mathsf{b}}\ell$**:** *lhs* $\equiv ((\$_{\mathsf{b}}^{\mathsf{n}\epsilon}\ell)(n{+}1))\{\theta_n\} = \mathsf{LIT}_{\mathsf{b}}(\ell)$. We have that $\mathcal{D}(\mathsf{LIT}_{\mathsf{b}}(\ell)) \equiv \ell \equiv |\mathcal{O}|$.

Case $\mathcal{O} \equiv v_i$**:** *lhs* $\equiv (v_i(n+1))\{\theta_n\} = \mathsf{VAR}(i)$. We have that $\mathcal{D}(\mathsf{VAR}(i)) \equiv v_i \equiv |\mathcal{O}|$.

Case $\mathcal{O} \equiv \underline{\lambda} x.\mathcal{O}_1$:

$$\begin{aligned} & lhs \equiv (\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda x.\{\!|\mathcal{O}_1|\!\}_{\mathsf{n}\epsilon})(n+1))\{\theta_n\} \\ = \;& \mathsf{LAM}(n+1, (\{\!|\mathcal{O}_1|\!\}_{\mathsf{n}\epsilon}\{\lambda i'.\mathsf{VAR}(n+1)/x\}((n+1)+1)))\{\theta_n\} \\ = \;& \mathsf{LAM}(n+1, (\{\!|\mathcal{O}_1\{v_{n+1}/x\}|\!\}_{\mathsf{n}\epsilon}((n+1)+1))\{\theta; \lambda i'.\mathsf{VAR}(n+1)/v_{n+1}\}) \\ \stackrel{i.h.}{=} \;& \mathsf{LAM}(n+1, t_1) \end{aligned}$$

where $\mathcal{D}(t_1) \sim_\alpha |\mathcal{O}_1|\{v_{n+1}/x\}$. Here we use the induction hypothesis on term $\mathcal{O}\{v_{n+1}/x\}$, which is typed as $v_1 : \bigcirc\sigma_1, \cdots, v_{n+1} : \bigcirc\sigma_{n+1} \rhd \mathcal{O}\{v_{n+1}/x\} : \bigcirc\sigma_{n+2}$, where $\sigma = \sigma_{n+1} \to \sigma_{n+2}$. We have that $\mathcal{D}(\mathsf{LAM}(n+1, t_1)) \equiv \lambda v_{n+1}.\mathcal{D}(t_1) \sim_\alpha \lambda x.|\mathcal{O}_1| \equiv |\mathcal{O}|$.

Case $\mathcal{O} \equiv \mathcal{O}_1 \underline{@} \mathcal{O}_2$:

$$\begin{aligned} & lhs \equiv (\underline{@}^{\mathsf{n}\epsilon}\{\!|\mathcal{O}_1|\!\}_{\mathsf{n}\epsilon}\{\!|\mathcal{O}_2|\!\}_{\mathsf{n}\epsilon}(n+1))\{\theta_n\} \\ = \;& \mathsf{APP}(\{\!|\mathcal{O}_1|\!\}_{\mathsf{n}\epsilon} n+1\{\theta_n\})\{\!|\mathcal{O}_2|\!\}_{\mathsf{n}\epsilon} n+1\{\theta_n\} \\ \stackrel{i.h.}{=} \;& \mathsf{APP}(t_1, t_2) \end{aligned}$$

where $\mathcal{D}(t_1) \sim_\alpha |\mathcal{O}_1|$ and $\mathcal{D}(t_2) \sim_\alpha |\mathcal{O}_2|$. We have that $\mathcal{D}(\mathsf{APP}(t_1, t_2)) = \mathcal{D}(t_1)\,\mathcal{D}(t_2) \sim_\alpha |\mathcal{O}_1|\,|\mathcal{O}_2| \sim_\alpha |\mathcal{O}|$

Case $\mathcal{O} \equiv \underline{d}$: $\quad lhs \equiv ((\lambda i.\mathsf{CST}(\wr d \int))(n+1))\{\theta_n\} = \wr d \int$. ☐

Lemma 2.9 (Computational adequacy). *If* $\mathsf{nPCF}^2 \vdash \rhd E : \bigcirc\sigma$, *and there is a* nPCF^{Λ}*-value* $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$, *then* $\exists\mathcal{O}.E \Downarrow \mathcal{O}$.

In the following, we write $E \Downarrow$ for $\exists V.E \Downarrow V$; that is, evaluation of E terminates.

We prove Lemma 2.9 using a Kripke logical relation between the denotation of translated terms and the original two-level terms.

Definition B.4 (Logical relation, $\prec_\tau^\Delta$). *For an object-type typing context* Δ, *we define, by induction on an* nPCF^2*-type* τ, *a family of relations* $v \prec_\tau^\Delta E$, *where* $v \in [\![\{\!|\tau|\!\}_{\mathsf{n}\epsilon}]\!]$, *and* $E \in Expr_\tau^\Delta = \{E \mid \mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau\}$, *by*

$$\begin{aligned} v \prec_{\mathsf{b}}^\Delta E &\iff v = \bot \lor \exists\ell.(v = \mathbf{val}^\bot\, \ell \land E \Downarrow \ell) \\ f \prec_{\bigcirc\sigma}^\Delta E &\iff \forall n.(f(\mathbf{val}^\bot\, n) = \bot \lor E \Downarrow) \\ f \prec_{\tau_1 \to \tau_2}^\Delta E &\iff \forall a \in [\![\{\!|\tau_1|\!\}_{\mathsf{n}\epsilon}]\!], \Delta' \geq \Delta, E' \in Expr_{\tau_1}^{\Delta'}. \\ & \qquad (a \prec_{\tau_1}^{\Delta'} E' \Rightarrow f(a) \prec_{\tau_2}^{\Delta'} EE') \end{aligned}$$

Note that the logical relation at the code types $\bigcirc\sigma$ only requires the termination of the evaluation of E. This requirement is enough for the proof, because programs cannot perform intensional analysis on values of types $\bigcirc\sigma$. In fact, it is essential for the correctness of the simple native embedding here that intensional analysis on code is absent from the language.

Before proceeding to prove a "Basic Lemma" for the logical relation, we first establish some of its properties.

"Weakening" holds for $\prec^{-}_{\tau}$ (Lemma B.5) Note that the object-type context Δ is used in the definition of the logical relation only to ensure that the term E is well-typed. Naturally we expect a weakening property; and indeed this property is used in several cases of the proof. It is this property that "forces" us to use a *Kripke-style* logical relation.

$\prec^{\Delta}_{\tau}$ is ω-admissible (Lemma B.6) This lemma is necessary for the proof in the case of a fixed-point operator.

Kleene equivalence respects $\prec^{\Delta}_{\tau}$ (Lemma B.7) At a few places, the operational semantics and the denotational semantics clash in a technical sense. The call-by-name denotational semantics, for example, does not use a lifted function space; the bottom element at type $\sigma_1 \to \sigma_2$ is not distinguished from a function value whose image (of type σ_2) is constantly bottom. In the operational semantics, the function needs to be evaluated to a value before the substitution. At base types, however, the two semantics agree.

The denotational semantics forces us to take a standard call-by-name form of logical relation at the function type in the definition. Another problem then appears: since we do not evaluate the expression form in the operational semantics, what we can infer from the induction hypothesis does not directly give the conclusion. In particular, for the case $\lambda x.E$, using the induction hypothesis, we can relate the denotation to the expression $E\{\theta; E'/x\}$, but instead we need to relate it to the expression $((\lambda x.E)\{\theta\})E'$. The two expressions evaluate to the same value in the operational semantics, i.e., they are Kleene-equivalent. Therefore, we need to show that the logical relation can be transferred between Kleene-equivalent terms.[11]

Lemma B.5 (Weakening of $\prec^{-}_{\tau}$). *If $\Delta \leq \Delta'$ and $v \prec^{\Delta}_{\tau} E$, then $v \prec^{\Delta'}_{\tau} E$.*

Proof. By a case analysis on type τ.

Case $\tau = \mathsf{b}$ or $\tau = \bigcirc\sigma$: Use the weakening property of the $\mathsf{nPCF^2}$ typing rules: $Expr^{\Delta}_{\tau} \subseteq Expr^{\Delta'}_{\tau}$.

Case $\tau = \tau_1 \to \tau_2$: Let $f \prec^{\Delta}_{\tau_1\to\tau_2} E$. First, we have that $E \in Expr^{\Delta}_{\tau} \subseteq Expr^{\Delta'}_{\tau}$. Second, let a, $\Delta'' \geq \Delta'$ (which implies $\Delta'' \geq \Delta$), and $E' \in Expr^{\Delta''}_{\tau_1}$ be such that $a \prec^{\Delta''}_{\tau_1} E'$. Then, by the definition of $\prec^{\Delta}_{\tau_1\to\tau_2}$, we have that $f(a) \prec^{\Delta''}_{\tau_2} EE'$. This shows that $f \prec^{\Delta'}_{\tau_1\to\tau_2} E$ by definition. □

[11] In the literature, this mismatch problem is resolved by using a different formulation of the logical relation, usually called *computability*: the relation at higher types is defined by means of full applications (e.g., Plotkin's proof of adequacy [46] and Gunter's proof [18, Section 4.3]), which reduces the definition at higher-type directly to ground type. The Kleene-equivalence formulation used in the present article has the same effect, but it seems to scale better with respect to other types such as product and sum; the definition of full applications becomes hairy in the presence of these types.

Lemma B.6 ($\prec_\tau^\Delta$ is ω-admissible). *For all $E \in Expr_\tau^\Delta$, the predicate $- \prec_\tau^\Delta E$ is admissible, i.e., (1) it is chain-complete: If $a_0 \sqsubseteq \ldots \sqsubseteq a_i \sqsubseteq \ldots$ is a countable chain in $[\![\{\!|\tau|\!\}_{\mathsf{n}\epsilon}]\!]$, such that $\forall i. a_i \prec_\tau^\Delta E$, then $\bigsqcup_{i\geq 0} a_i \prec_\tau^\Delta E$; (2) it is pointed: $\bot \prec_\tau^\Delta E$.*

Proof. By induction on type τ.

Case $\tau = \mathsf{b}$: Since $\{\!|\mathsf{b}|\!\}_{\mathsf{n}\epsilon} = \mathsf{b}$, and base types are interpreted by flat domains, the chain must be constant after a certain position, and the upper bound equals this constant. Pointedness follows from the definition of the logical relation.

Case $\tau = \bigcirc\sigma$: Let $f_0 \sqsubseteq \ldots \sqsubseteq f_i \sqsubseteq \ldots$ be a chain in $[\![\{\!|\tau|\!\}_{\mathsf{n}\epsilon}]\!] = \mathbf{Z}_\bot \to \mathbf{E}_\bot$. For a number n, if $(\bigsqcup_{i\geq 0} f_i)(\mathbf{val}^\bot\, n) = \bigsqcup_{i\geq 0}(f_i(\mathbf{val}^\bot\, n)) \neq \bot$, then $\exists m. f_m(\mathbf{val}^\bot\, n) \neq \bot$. This implies $E \Downarrow$, by the definition of $f_m \prec_{\bigcirc\sigma}^\Delta E$.

If $f = \bot$, the implication in the definition of the logical relation holds vacuously.

Case $\tau = \tau_1 \to \tau_2$: The predicate is given by

$$P(f) = \bigwedge_{(\Delta', a, E') \mid \Delta \leq \Delta', a \prec_{\tau_1}^{\Delta'} E'} (\lambda\phi.\phi a) f \prec_{\tau_2}^{\Delta'} EE'.$$

It is admissible since all the $- \prec_{\tau_2}^\Delta EE'$ are admissible by induction hypothesis, and admissibility is closed under taking pre-image under strict continuous function and arbitrary intersection [54]. $\square$

Lemma B.7 (Kleene equivalence respects $\prec_\tau^\Delta$). *If $v \prec_\tau^\Delta E$, and $E =^{\mathrm{kl}} E'$, i.e., $\forall V.(E \Downarrow V \Leftrightarrow E' \Downarrow V)$, then $v \prec_\tau^\Delta E'$.*

Proof. By induction on type τ.

Case $\tau = \mathsf{b}$ or $\tau = \bigcirc\sigma$: Immediate.

Case $\tau = \tau_1 \to \tau_2$: Let a, $\Delta' \geq \Delta$, and $E'' \in Expr_{\tau_1}^{\Delta'}$ be such that $a \prec_{\tau_1}^{\Delta'} E''$. Then we have

(1) $v(a) \prec_{\tau_2}^{\Delta'} EE''$ (by the definition of $v \prec_\tau^\Delta E$);

(2) $EE'' =^{\mathrm{kl}} E'E''$ (following from $E =^{\mathrm{kl}} E'$).

Applying the induction hypothesis for τ_2 to (1) and (2), we have that $v(a) \prec_{\tau_2}^{\Delta'} E'E''$. This shows that $v \prec_{\tau_1 \to \tau_2}^{\Delta'} E'$. $\square$

The logical relation extends naturally from types to typing contexts. Let Γ be a nPCF^2-typing context $(x_1 : \tau_1, \ldots, x_n : \tau_n)$, Δ an object-type typing context, and θ a substitution (from $\bigcirc\Delta$ to Γ) $\{E_1/x_1, \ldots, E_n/x_n\}$, where $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \triangleright$

$E_i : \tau_i$. We define the relation $\prec_\Gamma^\Delta$ between environments and substitutions as follows:

$$\rho \prec_\Gamma^\Delta \theta \iff \forall x \in dom\,\Gamma . \rho x \prec_{\Gamma(x)}^\Delta x\{\theta\}$$

Now we are ready to prove our version of the "Basic Lemma".

Lemma B.8. *Let* $\mathsf{nPCF}^2 \vdash \Gamma \rhd E : \tau$*, then for all* Δ*,* ρ*, and* θ*,* $\rho \prec_\Gamma^\Delta \theta$ *implies that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_\tau^\Delta E\{\theta\}$ *(it should be clear that* $\bigcirc\Delta \rhd E\{\theta\} : \tau$*).*

Proof. By induction on the size of the derivation for $\Gamma \rhd E : \tau$.

The static term formations have fairly standard sub-proofs.

Case ℓ: Trivial.

Case x: We need to show: $\rho x \prec_\tau^\Delta E\{\theta\}$. It follows from the definition of $\rho \prec_\Gamma^\Delta \theta$.

Case $\lambda x.E$: We need to show: $[\![\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\tau_1 \to \tau_2}^\Delta (\lambda x.E)\{\theta\}$. For all $\Delta' \geq \Delta$ and $a \prec_{\tau_1}^{\Delta'} E'$, we have $\rho \prec_\Gamma^{\Delta'} \theta$ by Lemma B.5, and therefore $\rho[x \mapsto a] \prec_{\Gamma, x:\tau_1}^{\Delta'} (\theta; E'/x)$. By induction hypothesis, we have

$$([\![\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho)a = [\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!](\rho[x \mapsto a]) \prec_{\tau_2}^{\Delta'} E\{\theta; E'/x\}$$

Since $((\lambda x.E)\{\theta\})E' =^{\mathrm{kl}} E\{\theta; E'/x\}$, we can apply Lemma B.7 to conclude that

$$([\![\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho)a \prec_{\tau_2}^\Delta ((\lambda x.E)\{\theta\})E'.$$

Case $E_1\, E_2$: We need to show: $[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}\, \{\!|E_2|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_\tau^\Delta (E_1\, E_2)\{\theta\}$. The induction hypotheses imply that $[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\tau_2 \to \tau}^\Delta E_1\{\theta\}$ and that $[\![\{\!|E_2|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\tau_2}^\Delta E_2\{\theta\}$. By definition of $\prec_{\tau_2 \to \tau}^\Delta$ (taking $\Delta' = \Delta$), we have

$$[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}\, \{\!|E_2|\!\}_{\mathsf{n}\epsilon}]\!]\rho = [\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}\rho]\!][\![\{\!|E_2|\!\}_{\mathsf{n}\epsilon}\rho]\!] \prec_\tau^\Delta (E_1\{\theta\})(E_2\{\theta\}) \equiv (E_1\, E_2)\{\theta\}$$

Case $\mathbf{fix}\, E$: We need to show: $\bigsqcup_{i \geq 0}([\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho)^i(\bot) \prec_\tau^\Delta (\mathbf{fix}\, E)\{\theta\}$. We have

$$\begin{array}{rcll} \bot & \prec_\tau^\Delta & (\mathbf{fix}\, E)\{\theta\} & (\textit{pointedness}) \\ ([\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho)(\bot) & \prec_\tau^\Delta & E\{\theta\}((\mathbf{fix}\, E)\{\theta\}) =^{\mathrm{kl}} (\mathbf{fix}\, E)\{\theta\} & (\textit{ind. hyp.}) \\ & \vdots & & \end{array}$$

By induction, for all $i \geq 0$, $([\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho)^i(\bot) \prec_\tau^\Delta (\mathbf{fix}\, E)\{\theta\}$. Finally, chain-completeness (Lemma B.6) implies the conclusion.

Case if $E_1\,E_2\,E_3$: We need to show:

$$[\![\mathbf{if}\ \{\!|E_1|\!\}_{\mathsf{n}\epsilon}\ \{\!|E_2|\!\}_{\mathsf{n}\epsilon}\ \{\!|E_3|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\tau}^{\Delta} \mathbf{if}\ E_1\ E_2\ E_3.$$

There are three sub-cases:

- $[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}]\!]\rho = \bot$: trivial.
- $[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}]\!]\rho = \mathbf{val}^{\bot}\ \mathsf{tt}$: the induction hypotheses imply that (1) $[\![\mathbf{if}\ \{\!|E_1|\!\}_{\mathsf{n}\epsilon}\ \{\!|E_2|\!\}_{\mathsf{n}\epsilon}\ \{\!|E_3|\!\}_{\mathsf{n}\epsilon}]\!]\rho = [\![\{\!|E_2|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\tau}^{\Delta} E_2\{\theta\}$, and (2) $E_1\{\theta\} \Downarrow \mathsf{tt}$, and henceforth $E_2\{\theta\} =^{\mathrm{kl}} (\mathbf{if}\ E_1\ E_2\ E_3)\{\theta\}$. Applying Lemma B.7 to (1) and (2), we have the conclusion.
- $[\![\{\!|E_1|\!\}_{\mathsf{n}\epsilon}]\!]\rho = \mathbf{val}^{\bot}\ \mathsf{ff}$: similar to the previous case.

Case $E_1 \otimes E_2$: Simple.

The sub-proofs for dynamic term formations are intuitively very simple: the denotational semantics of various constructs are strict in the sub-terms, which, by induction hypotheses, implies that the evaluation of the subterms terminates. However, in the case of a dynamic λ-abstraction $\underline{\lambda}x.E$, the change of typing context requires special attention to ensure that the term we use is well-typed. For other cases, we only show the proof for $\$_{\mathsf{b}}E$.

Case $\$_{\mathsf{b}}E$: We need to show: $[\![\$_{\mathsf{b}}^{\mathsf{n}\epsilon}\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\bigcirc\mathsf{b}}^{\Delta} (\$_{\mathsf{b}}E)\{\theta\}$. For $n \in \mathbf{Z}$, if $\bot \neq [\![\$_{\mathsf{b}}^{\mathsf{n}\epsilon}\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho(\mathbf{val}^{\bot}\ n) = [\![\mathsf{LIT}_{\mathsf{b}}(\{\!|E|\!\}_{\mathsf{n}\epsilon})]\!]\rho = \mathbf{let}^{\bot}\ l \Leftarrow [\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho\ \mathbf{in}\ \mathbf{val}^{\bot}\ (\mathit{inLit}_{\mathsf{b}}(l))$, then $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \neq \bot$. By induction hypothesis, we have $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec_{\mathsf{b}}^{\Delta} E\{\theta\}$, which implies that $E\{\theta\} \Downarrow$, and consequently $(\$_{\mathsf{b}}E)\{\theta\} \Downarrow$.

Case $\underline{\lambda}x.E$: Recall that the typing rule is

$$\frac{\Gamma, x{:}\bigcirc\sigma_1 \rhd E : \bigcirc\sigma_2}{\Gamma \rhd \underline{\lambda}x.E : \bigcirc(\sigma_1 \to \sigma_2)}$$

We need to show: $[\![\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon})]\!]\rho \prec_{\bigcirc(\sigma_1\to\sigma_2)}^{\Delta} (\underline{\lambda}x.E)\{\theta\}$. Without loss of generality, we assume $x \notin \mathit{dom}\,\Delta$; otherwise we can rename the bound variable using α-conversion. Now for any $n \in \mathbf{Z}$, if $[\![\underline{\lambda}^{\mathsf{n}\epsilon}(\lambda x.\{\!|E|\!\}_{\mathsf{n}\epsilon})]\!]\rho(\mathbf{val}^{\bot}\ n) \neq \bot$, then it is easy to show that $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho[x \mapsto \lambda w.\mathbf{val}^{\bot}\ (\mathit{inVar}(n))] \neq \bot$.

Since $x \notin \mathit{dom}\,\Delta$, the context $\Delta, x : \sigma_1$ is well-formed. It is easy to check that $\lambda w.\mathbf{val}^{\bot}\ (\mathit{inVar}(n)) \prec_{\bigcirc\sigma_1}^{\Delta,x:\sigma_1} x$; furthermore, since $\prec$ is Kripke, we also have that $\rho \prec_{\Gamma}^{\Delta,x:\sigma_1} \theta$. Putting them together, we have that

$$\rho[x \mapsto \lambda w.\mathbf{val}^{\bot}\ (\mathit{inVar}(n))] \prec_{\Gamma,x:\bigcirc\sigma_1}^{\Delta,x:\sigma_1} \{\theta; x/x\}.$$

Then, by the induction hypothesis, we get

$$[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!](\rho[x \mapsto \lambda w.\mathbf{val}^{\bot}\ (\mathit{inVar}(n))]) \prec_{\bigcirc\sigma_2}^{\Delta,x:\sigma_1} E\{\theta; x/x\} \equiv E\{\theta\}$$

Since $\mathit{lhs} \neq \bot$, we have, by the definition of the logical relation, that $E\{\theta\} \Downarrow$. Consequently $\underline{\lambda}x.(E\{\theta\}) \Downarrow$. □

Finally, Lemma 2.9 is an easy corollary.

Proof of Lemma 2.9. Let ρ be the empty environment, and θ the empty substitution. We have that $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!]\rho = [\![t]\!]\rho$. Since t is a value, $[\![t]\!]\rho \neq \perp$. Thus $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho(\mathbf{val}^{\perp}\ 1) = [\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!]\rho \neq \perp$. By Lemma B.8, $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec^{\cdot}_{\bigcirc\sigma} E\{\theta\} \equiv E$. The definition of the logical relation at the type $\bigcirc\sigma$ implies that $E\Downarrow$. $\square$

Theorem 2.10 (Correctness of embedding). *If* $\mathsf{nPCF}^{2} \vdash \rhd E : \bigcirc\sigma$*, then the following statements are equivalent.*
(a) There is a value $\mathcal{O} : \bigcirc\sigma$ *such that* $\mathsf{nPCF}^{2} \vdash E \Downarrow \mathcal{O}$*.*
(b) There is a value $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$*.*
When these statements hold, we further have that
(c) $\mathsf{nPCF} \vdash \rhd\mathcal{D}(t) : \sigma$ *and* $|\mathcal{O}| \sim_{\alpha} \mathcal{D}(t)$*.*

Proof. (a) $\Rightarrow$ (b),(c) Assume (a). By Lemma 2.7, $\mathsf{nPCF}^{\Lambda} \vdash \{\!|E|\!\}_{\mathsf{n}\epsilon} = \{\!|\mathcal{O}|\!\}_{\mathsf{n}\epsilon}$. By Lemma 2.8, there is a value t such that $\mathsf{nPCF}^{\Lambda} \vdash \{\!|\mathcal{O}|\!\}_{\mathsf{n}\epsilon}1 = t$, with which (c) also holds. By the transitivity rule, we have that $\mathsf{nPCF}^{\Lambda} \vdash \{\!|E|\!\}_{\mathsf{n}\epsilon}1 = t$. The conclusion now follows by an application of Theorem B.3.

(b) $\Rightarrow$ (a),(c) Assume (a). By Lemma 2.9, $E\Downarrow$. Since $\rhd E : \bigcirc\sigma$, we have $E \Downarrow \mathcal{O}$ for some value $\mathcal{O} : \bigcirc\sigma$. By the first part of this proof, we further have an nPCF^{Λ}-value $t' : \Lambda$ that satisfies (c) (with t replaced by t') and validates $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t']\!]$. It remains to show that $t \equiv t'$.

Because $[\![t]\!] = [\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t']\!]$, and the semantic function of Λ-typed values is injective (easy structural induction), we have that $t \equiv t'$.

$\square$

B.3 Call-by-name type-directed partial evaluation

Lemma 2.11. *For all types* σ*,* $\mathsf{nPCF} \vdash\ \rhd |\downarrow^{\sigma}| = \lambda x.x : \sigma \to \sigma$ *and* $\mathsf{nPCF} \vdash \rhd |\uparrow_{\sigma}| = \lambda x.x : \sigma \to \sigma$*.*

Proof. By a straightforward induction on type τ. $\square$

Theorem 2.12 (Semantic correctness of TDPE). *If* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$ *and* $\mathsf{nPCF}^{2} \vdash NF(E) \Downarrow \mathcal{O}$*, then* $\mathsf{nPCF} \vdash\ \rhd |\mathcal{O}| = |E| : \sigma$*.*

(Note that the two erasures are different: one operates on nPCF^{2}-terms, the other on $\mathsf{nPCF}^{\mathsf{tdpe}}$-terms.)

Proof. First, we prove by induction on $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \varphi$ that $\mathsf{nPCF} \vdash\ \rhd |\{\!|E|\!\}_{\mathsf{ri}}| = |E| : |\varphi|$. The proofs for the static part, for which both translations are homomorphic, are straightforward and omitted. The only remaining cases are the following ones.

Case $\$_{\mathsf{b}}E$: We have $|\{\!|\$_{\mathsf{b}}E|\!\}_{\mathsf{ri}}| \equiv |\$_{\mathsf{b}}\{\!|E|\!\}_{\mathsf{ri}}| \equiv |\{\!|E|\!\}_{\mathsf{ri}}| \stackrel{i.h.}{=} |E| \equiv |\$_{\mathsf{b}}E|$.

Case $d^{\mathfrak{d}} : \sigma^{\mathfrak{d}}$: We have $|\{\!|d^{\mathfrak{d}}|\!\}_{\mathsf{ri}}| \equiv |\uparrow_\sigma||\underline{d}| \stackrel{*}{=} d \equiv |d^{\mathfrak{d}}|$, where $\stackrel{*}{=}$ uses Lemma 2.11 and the definition of erasure for nPCF^2-terms.

From this, we can infer that $|NF(E)| \equiv |\downarrow^\sigma| \, |\{\!|E|\!\}_{\mathsf{ri}}| = |E|$, again using Lemma 2.11. Now, applying Theorem 2.3 to $\mathsf{nPCF}^2 \vdash NF(E) \Downarrow \mathcal{O}$, we can conclude that $|\mathcal{O}| = |NF(E)| = |E|$. $\square$

Theorem 2.13 (Refined type preservation). *If $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright E : \tau$ and $\mathsf{nPCF}^2 \vdash E \Downarrow V$, then $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \tau$.*

Like in the proof of Theorem 2.1, the most interesting case is when $E \equiv E_1\, E_2$, for which we need a Substitution Lemma.

Lemma B.9 ($\blacktriangleright$-substitutivity). *If $\mathsf{nPCF}^2 \vdash \Gamma, y : \tau_1 \blacktriangleright E : \tau_2$ and $\mathsf{nPCF}^2 \vdash \Gamma \blacktriangleright E' : \tau_1$, then $\mathsf{nPCF}^2 \vdash \Gamma \blacktriangleright E\{E'/y\} : \tau_2$.*

Proof. By a straightforward induction on the derivation of the typing judgment $\mathsf{nPCF}^2 \vdash \Gamma, x : \tau_1 \blacktriangleright E : \tau_2$. $\square$

Proof of Theorem 2.13. By induction on $E \Downarrow V$. The only non-straightforward case in the static part is the rule ($[\mathit{app}]$), i.e., the evaluation of an application; for this rule we use Lemma B.9. For the dynamic part, all the rules simply evaluate the subterms while keeping the top-level constructs; combining the induction hypotheses suffices to give the same typing for the result values V as for the original terms E. $\square$

Theorem 2.14 (Normal-form code types). *If V is an nPCF^2-value (Figure 2), then*
(1) if $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \bigcirc^{at}(\sigma)$, then $V \equiv \mathcal{O}$ for some $\mathcal{O}$ and $\Delta \rhd^{at} |\mathcal{O}| : \sigma$;
(2) if $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \bigcirc^{nf}(\sigma)$, then $V \equiv \mathcal{O}$ for some $\mathcal{O}$ and $\Delta \rhd^{nf} |\mathcal{O}| : \sigma$.

Proof. First of all, if a value V is of any code type $\bigcirc^{var}(\sigma)$, $\bigcirc^{at}(\sigma)$, or $\bigcirc^{nf}(\sigma)$, then a simple examination of the rules shows that V can be neither of the form ℓ nor of the form $\lambda x.E$, and thus it must be of the form $\mathcal{O}$. Furthermore, if $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright \mathcal{O} : \bigcirc^{var}(\sigma)$, then $\mathcal{O}$ must be a variable x such that $x : \sigma \in \Delta$, since in all other cases of $\mathcal{O}$, the type could not be $\bigcirc^{var}(\sigma)$.

According to the BNF for a code-typed value $\mathcal{O}$, the only rules that can be used in the derivation of $\mathcal{O}$'s typing are the rules in the (new) dynamic part plus the rules for literals and variables. Now, a simple rule induction proves (1) and (2). $\square$

Lemma 2.15. *(1) The extraction functions (Figure 5c) have the following normal-form types (writing $\sigma^{\bigcirc \mathsf{nf}}$ for $\sigma\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$).*

$$\blacktriangleright \downarrow^\sigma : \sigma^{\bigcirc \mathsf{nf}} \to \bigcirc^{nf}(\sigma), \blacktriangleright \uparrow_\sigma : \bigcirc^{at}(\sigma) \to \sigma^{\bigcirc \mathsf{nf}}.$$

(2) If $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi$, then $\mathsf{nPCF}^2 \vdash \{\!|\Gamma|\!\}^{\mathsf{nf}}_{\mathsf{ri}} \blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}^{\mathsf{nf}}_{\mathsf{ri}}$, where $\{\!|\varphi|\!\}^{\mathsf{nf}}_{\mathsf{ri}} = \varphi\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b}^{\mathfrak{d}} : \mathsf{b} \in \mathbb{B}\}$

Proof.
(1) By induction on type σ.

Case $\sigma = \mathsf{b}$: Because at the base type, $\mathsf{b}^{\bigcirc \mathsf{nf}} = \bigcirc^{nf}(\mathsf{b})$, we just need to show: $\blacktriangleright \lambda x.x : \bigcirc^{nf}(\mathsf{b}) \to \bigcirc^{nf}(\mathsf{b})$. This is simple.

Case $\sigma = \sigma_1 \to \sigma_2$: Noting that $(\sigma_1 \to \sigma_2)^{\bigcirc \mathsf{nf}} = \sigma_1^{\bigcirc \mathsf{nf}} \to \sigma_2^{\bigcirc \mathsf{nf}}$, we give the following typing derivation for $\downarrow^{\sigma_1 \to \sigma_2}$ based on the induction hypotheses (we use weakening freely and implicitly, write Γ_1 for the context $f : \sigma_1^{\bigcirc \mathsf{nf}} \to \sigma_2^{\bigcirc \mathsf{nf}}, x : \bigcirc^{var}(\sigma_1)$, and omit the typing of $\uparrow_{\sigma_1}$ and $\downarrow^{\sigma_2}$ from the induction hypotheses):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \blacktriangleright f : \sigma_1^{\bigcirc \mathsf{nf}} \to \sigma_2^{\bigcirc \mathsf{nf}} \qquad \dfrac{\dfrac{\Gamma_1 \blacktriangleright x : \bigcirc^{var}(\sigma_1)}{\Gamma_1 \blacktriangleright x : \bigcirc^{at}(\sigma_1)}}{\Gamma_1 \blacktriangleright \uparrow_{\sigma_1} x : \sigma_1^{\bigcirc \mathsf{nf}}}}{\Gamma_1 \blacktriangleright f\,(\uparrow_{\sigma_1} x) : \sigma_2^{\bigcirc \mathsf{nf}}}}{\Gamma_1 \blacktriangleright \downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x)) : \bigcirc^{nf}(\sigma_2)}}{f : \sigma_1^{\bigcirc \mathsf{nf}} \to \sigma_2^{\bigcirc \mathsf{nf}} \blacktriangleright \underline{\lambda} x.\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x)) : \bigcirc^{nf}(\sigma_1 \to \sigma_2)}}{\blacktriangleright \lambda f.\underline{\lambda} x.\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x)) : (\sigma_1^{\bigcirc \mathsf{nf}} \to \sigma_2^{\bigcirc \mathsf{nf}}) \to \bigcirc^{nf}(\sigma_1 \to \sigma_2)}$$

A similar derivation works for $\uparrow_{\sigma_1 \to \sigma_2}$, which is compactly described as the following:

$$e : \bigcirc^{at}(\sigma_1 \to \sigma_2), x : \sigma_1^{\bigcirc \mathsf{nf}} \blacktriangleright \uparrow_{\sigma_2}(\overbrace{e \,\underline{@}\, (\underbrace{\downarrow^{\sigma_1} x}_{\bigcirc^{nf}(\sigma_1)})}^{\bigcirc^{at}(\sigma_2)}) : \sigma_2^{\bigcirc \mathsf{nf}}$$

(2) By a simple induction on $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi$. For the case when $E \equiv d^{\mathfrak{d}}$ with $Sg(d) = \sigma$, we use the typing of $\uparrow_\sigma$ from part (1) and the fact that $\{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nf}}_{\mathsf{ri}} \equiv \{\!|\sigma\{\mathsf{b}^{\mathfrak{d}}/\mathsf{b} : b \in \mathbb{B}\}|\!\}^{\mathsf{nf}}_{\mathsf{ri}} \equiv \sigma\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\} \equiv \sigma^{\bigcirc \mathsf{nf}}$. □

Theorem 2.16. *If $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, then $\mathsf{nPCF}^{2} \vdash\blacktriangleright NF(E) : \bigcirc^{nf}(\sigma)$.*

Proof. By Lemma 2.15(2), we have $\mathsf{nPCF}^{2} \vdash\blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nf}}_{\mathsf{ri}}$. Since $\{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nf}}_{\mathsf{ri}} \equiv \sigma^{\bigcirc \mathsf{nf}}$, applying $\downarrow^{\sigma} : \sigma^{\bigcirc \mathsf{nf}} \to \bigcirc^{nf}(\sigma)$ (Lemma 2.15(1)) to $\{\!|E|\!\}_{\mathsf{ri}}$ yields the conclusion. □

Corollary 2.17 (Syntactic correctness of TDPE). *For $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, if $\mathsf{nPCF}^{2} \vdash NF(E) \Downarrow V$, then $V \equiv \mathcal{O}$ for some $\mathcal{O}$ and $\mathsf{nPCF} \vdash \Delta \rhd^{nf} |\mathcal{O}| : \sigma$.*

Proof. We use Theorem 2.16, Theorem 2.13, and Theorem 2.14. □

C Call-by-value two-level language vPCF^{2}: detailed development

Since we are working with vPCF^{2} in this section, we leave $\mathsf{vPCF}^{2} \vdash$ implicit.

C.1 Type preservation

Lemma C.1 (Substitution Lemma for vPCF²-typing). *If* $\Gamma, y : \tau_1 \triangleright E : \tau_2$ *and* $\Gamma \triangleright E' : \tau_1$*, then* $\Gamma \triangleright E\{E'/y\} : \tau_2$*.*

For a concise presentation, we introduce the following notations.

Definition C.2 (Binder-in-context). *For a* binder $B \equiv (x_1{:}\sigma_1 = \mathcal{O}_1, \ldots, x_n{:}\sigma_n = \mathcal{O}_n)$*, we write* $\Gamma \triangleright [B]$ *if* $\Gamma, x_1 : ⓥ\sigma_1, \ldots, x_{i-1} : ⓥ\sigma_{i-1} \triangleright \mathcal{O}_i : ⓔ\sigma_i$ *for all* $1 \le i \le n$*. In this case, we also write* $\Gamma, ⓥB$ *for the context* $\Gamma, ⓥ\sigma_1, \ldots, ⓥ\sigma_n$*.*

Definition C.3 (Binder extension and difference). *We write* $\Gamma \triangleright [B] \ge [B']$*, if binder* B' *is a prefix of binder* B*, and* $\Gamma \triangleright [B]$*. In this case, it makes sense to write* $B - B'$ *to denote the difference of the two binders, and we have that* $\Gamma, ⓥB \triangleright [B - B']$*.*

We can restate Definition 4.1 using the preceding notions.

Definition 4.1 (Binder-term-in-context). *We write* $\Gamma \triangleright [B]E : \tau$ *if* $\Gamma \triangleright [B]$ *and* $\Gamma, ⓥB \triangleright E : \tau$*.*

To ease analyses of binder-terms-in-context, we prove an "inversion" lemma. Informally, it states that one can apply inversion to a binder-term-in-context as if to a term-in-context,

Lemma C.4 ("Inversion" holds for binder-terms-in-context). *Let* $\Gamma \triangleright [B]E : \tau$*.*

- *If* $E = \lambda x.E'$*, then* $\tau = \tau_1 \to \tau_2$ *and* $\Gamma, x : \tau_1 \triangleright [B]E' : \tau_2$*. This corresponds to the typing rule* [*lam*]*.*
- *If* $E = E_1\, E_2$*, then* $\exists \tau_2$ *such that* $\Gamma \triangleright [B]E_1 : \tau_2 \to \tau$ *and* $\Gamma \triangleright [B]E_2 : \tau_2$*. This corresponds to the typing rule* [*app*]*.*
- $\vdots$ *(similar inversion principles for all the other typing rules, except the rule* [*var*]*, which uses the context explicitly.)*

Corollary C.5. *(Substitution for binder-term-in-context) If* $\Gamma, y : \tau_1 \triangleright [B]E : \tau_2$ *and* $\Gamma \triangleright [B]E' : \tau_1$*, then* $\Gamma \triangleright [B]E\{E'/x\} : \tau_2$*.*

Proof. The provability of binder-term-in-context $\Gamma \triangleright [B]E' : \tau_1$, by definition, implies the following.

(1.a) $\Gamma \triangleright [B]$.

(1.b) $\Gamma, ⓥB \triangleright E' : \tau_1$.

From (1.a) and $\Gamma, y : \tau_1 \triangleright [B]E : \tau_2$ we have

(2) $(\Gamma, ⓥB), y : \tau_1 \triangleright E : \tau_2$.

Applying Lemma C.1 to (1.b) and (2) yields the conclusion. □

Lemma C.6. *If* $\Gamma \rhd [B]E : \tau$ *and* $\Gamma \rhd [B'] \geq [B]$*, then* $\Gamma \rhd [B']E : \tau$*.*

Proof. The definition of $\Gamma \rhd [B]E : \tau$ implies

(1) $\Gamma, ⓥB \rhd E : \tau$.

The definition of $\Gamma \rhd [B'] \geq [B]$ implies

(2.a) $\Gamma \rhd [B']$.

(2.b) $\Gamma, ⓥB \subseteq \Gamma, ⓥB'$.

Weakening (1) with respect to (2.b) yields $\Gamma, ⓥB' \rhd E : \tau$, which, along with (2.a), is exactly the definition of $\Gamma \rhd [B']E : \tau$. □

If a binder-term-in-context has a code type ⓔτ, then we can convert it into a term-in-context using the let-construct. This observation is manifested in the following lemma.

Lemma C.7. *If* $\Gamma \rhd [B]\mathcal{O} :$ ⓔτ*, then* $\Gamma \rhd \mathbf{let}^* \ B \ \mathbf{in} \ \mathcal{O} :$ ⓔτ*.*

We slightly strengthen the type preservation theorem 4.2, so as to make the induction go through.

Theorem 4.2 (Type preservation). *If* ⓥ$\Delta \rhd [B]E : \tau$ *and* $\Delta \rhd [B]E \Downarrow [B']V$*, then (1)* ⓥ$\Delta \rhd [B']V : \tau$*, and (2)* ⓥ$\Delta \rhd [B'] \geq [B]$*.*

Proof. By induction on $\Delta \rhd [B]E \Downarrow [B']V$. For (2), note that, for all the implicit rules, $\Delta \rhd [B_{n+1}] \geq \ldots \geq [B_1]$ follows immediately from transitivity of "$\geq$" and the induction hypotheses; for the three rules where binders are explicitly mentioned, it is also clear that (2) holds.

For (1), we show a few cases.

Case [*lit*]**,** [*lam*]**:** There is nothing to prove.

Case [*app*]**:** The rule in its full from is

$$\frac{\Delta \rhd [B_1]E_1 \Downarrow [B_2]\lambda x.E' \quad \Delta \rhd [B_2]E_2 \Downarrow [B_3]V' \quad \Delta \rhd [B_3]E'\{V'/x\} \Downarrow [B_4]V}{\Delta \rhd [B_1]E_1 \ E_2 \Downarrow [B_4]V}$$

for which we reason as follows:

(1) By assumption, ⓥ$\Delta \rhd [B_1]E_1 \ E_2 : \tau$.

(2) Inverting (Lemma C.4) (1) gives

a. ⓥ$\Delta \rhd [B_1]E_1 : \theta_2 \to \tau$, and

b. ⓥ$\Delta \rhd [B_1]E_2 : \theta_2$.

(3) From (2.a), by induction hypothesis 1 (counting from left to right),

a. $\textcircled{v}\Delta \rhd [B_2]\lambda x.E' : \theta_2 \to \tau$, and
b. $\textcircled{v}\Delta \rhd [B_2] \geq [B_1]$.

(4) Applying Lemma C.6 to (2.b) and (3.b) yields

$$\textcircled{v}\Delta \rhd [B_2]E_2 : \theta_2$$

By induction hypothesis 2,

a. $\textcircled{v}\Delta \rhd [B_3]V' : \theta_2$
b. $\textcircled{v}\Delta \rhd [B_3] \geq [B_2]$.

(5) Applying Lemma C.6 to (4.b) and (3.a), and then inversion yields

$$\textcircled{v}\Delta, x : \theta_2 \rhd [B_3]E' : \tau$$

(6) Applying Substitution (Corollary C.5) to (5) and (4.a) yields

$$\textcircled{v}\Delta \rhd [B_3]E'\{V'/x\} : \tau.$$

By induction hypothesis 3,

$$\textcircled{v}\Delta \rhd [B_4]V : \tau.$$

Case [*fix*]**:** The rule in its full form is

$$\frac{\Delta \rhd [B_1]E \Downarrow [B_2]\lambda x.E' \quad \Delta \rhd [B_2]E'\{\mathbf{fix}\,(\lambda x.E')/x\} \Downarrow [B_3]V}{\Delta \rhd [B_1]\mathbf{fix}\,E \Downarrow [B_3]V}$$

for which we reason as follows:

(1) By assumption, $\textcircled{v}\Delta \rhd [B_1]\mathbf{fix}\,E : \tau$ (where $\tau = \theta_1 \to \tau_2$ for some θ_1 and τ_2).

(2) Inverting (1) gives

$$\textcircled{v}\Delta \rhd [B_1]E : \tau \to \tau.$$

(3) Applying induction hypothesis 1 to (2) yields

$$\textcircled{v}\Delta \rhd [B_2]\lambda x.E' : \tau \to \tau.$$

(4) Inverting (3) gives

a. $\textcircled{v}\Delta, x : \tau \rhd [B_2]E' : \tau$

An application of the typing rule ([*fix*]) to (3) gives

b. $\textcircled{v}\Delta \rhd [B_2]\mathbf{fix}\,\lambda x.E' : \tau$

(5) Applying Substitution to (4.a) and (4.b) yields

$$\textcircled{v}\Delta \rhd [B_2]E'\{\mathbf{fix}\,(\lambda x.E')/x\} : \tau.$$

(6) Applying induction hypothesis 2 to (5) yields

$$\textcircled{v}\Delta \rhd [B_3]V : \tau.$$

Case [$\underline{\mathit{lam}}$]**:** The rule is

$$\frac{\Delta, y : \sigma, B \rhd [\cdot]E\{y/x\} \Downarrow [B']\mathcal{O} \quad y \notin \mathit{dom}\, B \cup \mathit{dom}\, \Delta}{\Delta \rhd [B]\underline{\lambda}x.E \Downarrow [B]\underline{\lambda}y.\underline{\mathbf{let}}^{*}\ B'\ \underline{\mathbf{in}}\ \mathcal{O}}$$

for which we reason as follows:

(1) By assumption, $\textcircled{v}\Delta \rhd [B]\underline{\lambda}x.E : \textcircled{e}(\sigma_1 \to \sigma_2)$

(2) Inverting (1) gives

$$\textcircled{v}\Delta, x : \textcircled{v}\sigma_1 \rhd [B]E : \textcircled{e}\sigma_2$$

from which it follows that $\textcircled{v}\Delta, y : \textcircled{v}\sigma_1 \rhd [B]E\{y/x\} : \textcircled{e}\sigma_2$. That is (noting that $\textcircled{v}\Delta, y : \textcircled{v}\sigma_1 \equiv \textcircled{v}(\Delta, y : \sigma_1)$)

a. $\textcircled{v}(\Delta, y : \sigma_1) \rhd [B]$, and

b. $\textcircled{v}(\Delta, y : \sigma_1, B) \rhd E\{y/x\} : \textcircled{e}\sigma_2$.

(3) Use the induction hypothesis on (2.b) (noting that $\Gamma \rhd E : \tau \Leftrightarrow \Gamma \rhd [\cdot]E : \tau$), we have

$$\textcircled{v}(\Delta, y : \sigma_1, B) \rhd [B']\mathcal{O} : \textcircled{e}\sigma_2$$

(4) Apply Lemma C.7 to (3) yields

$$\textcircled{v}(\Delta, y : \sigma_1, B) \rhd \mathbf{let}^{*}\ B'\ \mathbf{in}\ \mathcal{O} : \textcircled{e}\sigma_2$$

Finally, applying the typing rule for dynamic lambda $\underline{\lambda}y.E$, [$\underline{\mathit{lam}}$], yields $\textcircled{v}\Delta, \textcircled{v}B \rhd \underline{\lambda}y.\mathbf{let}^{*}\ B'\ \mathbf{in}\ \mathcal{O} : \textcircled{e}(\sigma_1 \to \sigma_2)$.

Case [$\underline{\mathit{let}}$]**:** Similar to the case of rule [$\underline{\mathit{lam}}$].

Case [#]**:** The rule is

$$\frac{\Delta \rhd [B]E \Downarrow [B']\mathcal{O} \quad x \notin \mathit{dom}\, B' \cup \mathit{dom}\, \Delta}{\Delta \rhd [B]\#E \Downarrow [B', x : \sigma = \mathcal{O}]x}$$

for which we reason as follows:

(1) By assumption, $\textcircled{v}\Delta \rhd [B]\#E : \tau$. Without loss of generality, we can assume that $\tau = \textcircled{v}\sigma$; the other case, where $\tau = \textcircled{e}\sigma$, can be reduced to this case using one inversion. Therefore, we have,

$$\textcircled{v}\Delta \rhd [B]\#E : \textcircled{v}\sigma.$$

(2) Inverting (1) gives

$$\textcircled{v}\Delta \rhd [B]E : \textcircled{e}\sigma$$

(3) Use induction hypothesis on (2) gives

$$ⓥ\Delta \rhd [B']\mathcal{O} : ⓔ\sigma.$$

Because $x \notin dom\, B' \cup dom\, \Delta$, we have $ⓥ\Delta \rhd [B', x : \sigma = \mathcal{O}]$ by definition. We also have $ⓥ(\Delta, B), x : ⓥ\sigma \rhd x : ⓥ\sigma$. Therefore, by definition, we have

$$ⓥ\Delta \rhd [B', x : \sigma = \mathcal{O}]x : ⓥ\sigma.$$

$\square$

Recall that the observation of a complete program is defined to be that of code type.

Definition 4.3 (Observation of complete program). *For a complete program $\rhd E : ⓔ\sigma$, we write $E \searrow \underline{\mathbf{let}}^* \ B \ \underline{\mathbf{in}} \ \mathcal{O}$ if $\rhd[\cdot]E \Downarrow [B]\mathcal{O}$.*

In accordance with this understanding of complete-program semantics, the following corollary of the Type Preservation theorem (Theorem 4.2) provides type preservation for a complete program.

Corollary 4.4 (Type preservation for complete programs). *If $\rhd E : ⓔ\sigma$ and $E \searrow \mathcal{O}$, then $\rhd\mathcal{O} : ⓔ\sigma$.*

Proof. Assume $\rhd[\cdot]E \Downarrow [B]\mathcal{O}'$ where $\mathcal{O} \equiv \mathbf{let}^* \ B \ \mathbf{in} \ \mathcal{O}'$. We have

$$\begin{array}{ll} & \rhd E : ⓔ\sigma \\ \Longrightarrow & \rhd[\cdot]E : ⓔ\sigma \\ \stackrel{(*)}{\Longrightarrow} & \rhd[B]\mathcal{O}' : ⓔ\sigma \\ \stackrel{(**)}{\Longrightarrow} & \rhd\mathbf{let}^* \ B \ \mathbf{in} \ \mathcal{O}' : ⓔ\sigma \end{array}$$

where $(*)$ follows from Theorem 4.2 and $(**)$ follows from Lemma C.7. $\square$

C.2 Determinacy

Next, we would like to show that the operational semantics of vPCF² is deterministic. At the top level, determinacy can be easily phrased as "evaluating a whole program gives a unique result, modulo α-equivalence". Going into the inductive steps for the proof, however, requires an extension of the notion of α-equivalence to take into account of binders, and in turn, of contexts. This extra conceptual complexity is induced, in particular, by the explicit name generations and context manipulation in rules such as [$\underline{lam}$].

Definition C.8 (Name substitution). *A name substitution is a substitution that maps variable names to variable names. Applying a name substitution θ to a context Γ substitutes the variable names in Γ:*

$$\Gamma\{\theta\} \triangleq \{(x\{\theta\} : \tau) | (x : \tau) \in \Gamma\}$$

Definition C.9 (α-equivalence for terms-in-context). *Let $\Gamma \rhd E : \tau$ and $\Gamma' \rhd E' : \tau$ be two valid terms-in-context. We say that they are α-equivalent, noted as $(\Gamma \rhd E) \sim_\alpha (\Gamma' \rhd E')$, if there exists a name substitution θ such that*

$$\begin{array}{rcl} \Gamma' & \equiv & \Gamma\{\theta\} \\ E' & \sim_\alpha & E\{\theta\} \end{array}$$

We then define the corresponding notion of *α-equivalence for binders-in-context* (omitted, cf. Definition C.2), and the subsequent notion of *α-equivalence for binder-terms-in-context.*

Definition C.10 (α-equivalence for binder-terms-in-context). *Let $\Gamma \rhd [B]E : \tau$ and $\Gamma' \rhd [B']E' : \tau$. We say that they are α-equivalent if $(\Gamma \rhd [B]) \sim_\alpha (\Gamma' \rhd [B'])$ and $(\Gamma, ⓥB \rhd E) \sim_\alpha (\Gamma', ⓥB' \rhd E')$.*

All the above notions of α-equivalences define equivalence relations. This follows immediately from the fact that the usual α-equivalence defines a equivalence relation.

Lemma C.11 (Collecting binders preserves α-equivalence). *Let $\Gamma \rhd [B]E : \bigcirc\sigma$ and $\Gamma' \rhd [B']E' : \bigcirc\sigma$ be two valid terms-in-contexts. If $(\Gamma \rhd [B]E) \sim_\alpha (\Gamma' \rhd [B']E')$, then $(\Gamma \rhd \underline{\mathbf{let}}^*\ B\ \underline{\mathbf{in}}\ E) \sim_\alpha (\Gamma' \rhd \underline{\mathbf{let}}^*\ B'\ \underline{\mathbf{in}}\ E')$.*

Proof. Immediate. □

Lemma C.12. *Let $(\Gamma \rhd [B]\lambda x.E) \sim_\alpha (\Gamma' \rhd [B']\lambda x'.E')$ and $(\Gamma \rhd [B]E_1) \sim_\alpha (\Gamma' \rhd [B']E_1')$. Then, $(\Gamma \rhd [B]E\{E_1/x\}) \sim_\alpha (\Gamma' \rhd [B']E'\{E_1'/x\})$.*

Theorem C.13 (Determinacy modulo α-conversion). *Let $ⓥ\Delta \rhd [B_1]E_1 : \tau$ and $ⓥ\Delta' \rhd [B_1']E_1' : \tau$ be valid terms-in-contexts such that $(ⓥ\Delta \rhd [B_1]E_1) \sim_\alpha (ⓥ\Delta' \rhd [B_1']E_1')$. If $\Delta \rhd [B_1]E_1 \Downarrow [B_2]E_2$, then (1) for all B_2' and E_2' such that $\Delta' \rhd [B_1']E_1' \Downarrow [B_2']E_2'$, we have $(ⓥ\Delta \rhd [B_2]E_2) \sim_\alpha (ⓥ\Delta' \rhd [B_2']E_2')$; and (2) such B_2' and E_2' exist.*

Furthermore, the derivation trees for $\Delta \rhd [B_1]E_1 \Downarrow [B_2]E_2$ and for $\Delta' \rhd [B_1']E_1' \Downarrow [B_2']E_2'$ have exactly the same shape.

Proof. By induction on $\Delta \rhd [B_1]E_1 \Downarrow [B_2]E_2$, we prove that for all such Δ', B_1', E_1', B_2', E_2' that satisfy the rest of the premises, it holds that $(ⓥ\Delta_1 \rhd [B_1']E_1') \sim_\alpha (ⓥ\Delta_2 \rhd [B_2']E_2')$. The proof for (2) is easy, so we concentrate on (1).

We demonstrate two cases. For the expression forms E' that have a unique inversions (i.e., those that are not **if**-expressions), we omit the inversion of E'.

Case [*app*]: The derivations end with

$$\frac{\Delta \rhd [B_1]E_1 \Downarrow [B_2]\lambda x.E_3 \quad \Delta \rhd [B_2]E_2 \Downarrow [B_3]V_1 \quad \Delta \rhd [B_3]E_3\{V_1/x\} \Downarrow [B_4]V}{\Delta \rhd [B_1]E_1\ E_2 \Downarrow [B_4]V}$$

for which we have the following reasoning:

(1) The assumption $(ⓥ\Delta \rhd [B_1]E_1\ E_2) \sim_\alpha (ⓥ\Delta' \rhd [B_1']E_1'\ E_2')$ implies

a. $(ⓥ\Delta \rhd [B_1]E_1) \sim_\alpha (ⓥ\Delta' \rhd [B'_1]E'_1)$, and

b. $(ⓥ\Delta \rhd [B_1]E_2) \sim_\alpha (ⓥ\Delta' \rhd [B'_1]E'_2)$.

(2) From (1.a), by induction hypothesis 1:

$$(ⓥ\Delta \rhd [B_2]\lambda x.E_3) \sim_\alpha (ⓥ\Delta' \rhd [B'_2]\lambda x.E'_3).$$

(3) From (1.b) and that binders B_2 and B'_2 extend B_1 and B'_1, by induction hypothesis 2:

$$(ⓥ\Delta \rhd [B_3]V_1) \sim_\alpha (ⓥ\Delta' \rhd [B'_3]V'_1)$$

(4) An application of Lemma C.12 to (2) and (3) (with binders properly extended), followed by an application of hypothesis 3, gives the conclusion.

Case [*lam*]**:** The derivations end with

$$\frac{\Delta, y:\sigma; B_1 \rhd [\cdot]E\{y/x\} \Downarrow [B_2]\mathcal{O} \quad y \notin \mathit{dom}\, B_1 \cup \mathit{dom}\, \Delta}{\Delta \rhd [B_1]\underline{\lambda} x.E \Downarrow [B_1]\underline{\lambda} y.\underline{\mathbf{let}}^*\ B_2\ \underline{\mathbf{in}}\ \mathcal{O}}$$

for which we have the following reasoning:

(1) The assumption $(ⓥ\Delta \rhd [B_1]\lambda x.E) \sim_\alpha (ⓥ\Delta' \rhd [B'_1]\lambda x'.E')$ implies that

$$(ⓥ\Delta, ⓥB_1 \rhd \lambda x.E) \sim_\alpha (ⓥ\Delta', ⓥB'_1 \rhd \lambda x'.E')$$

which, by definition, implies that

$$(ⓥ(\Delta, y:\sigma_1), ⓥB_1 \rhd E\{y/x\}) \sim_\alpha (ⓥ(\Delta', y':\sigma_1), ⓥB'_1 \rhd E'\{y'/x'\}).$$

(2) An application of the induction hypothesis to (1) yields

$$(ⓥ(\Delta, y:\sigma_1), ⓥB_1 \rhd [B_2]\mathcal{O}) \sim_\alpha (ⓥ(\Delta', y':\sigma_1), ⓥB'_1 \rhd [B'_2]\mathcal{O}')$$

By Lemma C.11, we have

$$(ⓥ(\Delta, y:\sigma_1), ⓥB_1 \rhd \underline{\mathbf{let}}^*\ B_2\ \underline{\mathbf{in}}\ \mathcal{O})$$

$$\sim_\alpha$$

$$(ⓥ(\Delta', y':\sigma'_1), ⓥB'_1 \rhd \underline{\mathbf{let}}^*\ B'_2\ \underline{\mathbf{in}}\ \mathcal{O}')$$

This implies the conclusion.

□

Types $\sigma ::= \mathsf{b} \mid \sigma_1 \to \sigma_2$

Raw terms $E ::= \ell \mid x \mid d \mid \lambda x.E \mid E_1\,E_2 \mid \mathbf{fix}\,E$
$\mid \mathbf{if}\,E_1\,E_2\,E_3 \mid E_1 \otimes E_2 \mid \mathbf{let}\ x \Leftarrow E_1\ \mathbf{in}\ E_2$

Typing Judgment $\boxed{\mathsf{vPCF} \vdash \Delta \rhd E : \sigma}$

The typing rules are very close to that of nPCF. For the convenience of the reader, we display the complete rules here.

$$[\mathit{lit}]\,\frac{\ell \in \mathbb{L}(\mathsf{b})}{\Delta \rhd \ell : \mathsf{b}} \quad [\mathit{var}]\,\frac{x{:}\,\sigma \in \Delta}{\Delta \rhd x : \sigma} \quad [\mathit{cst}]\,\frac{Sg(d) = \sigma}{\Delta \rhd d : \sigma} \quad [\mathit{lam}]\,\frac{\Delta, x{:}\,\sigma_1 \rhd E : \sigma_2}{\Delta \rhd \lambda x.E : \sigma_1 \to \sigma_2}$$

$$[\mathit{app}]\,\frac{\Delta \rhd E_1 : \sigma_2 \to \sigma \quad \Delta \rhd E_2 : \sigma_2}{\Delta \rhd E_1\,E_2 : \sigma} \quad [\mathit{fix}]\,\frac{\Delta \rhd E : (\sigma_1 \to \sigma_2) \to (\sigma_1 \to \sigma_2)}{\Delta \rhd \mathbf{fix}\,E : \sigma_1 \to \sigma_2}$$

$$[\mathit{if}]\,\frac{\Delta \rhd E_1 : \mathsf{bool} \quad \Delta \rhd E_2 : \sigma \quad \Delta \rhd E_3 : \sigma}{\Delta \rhd \mathbf{if}\,E_1\,E_2\,E_3 : \sigma}$$

$$[\mathit{bop}]\,\frac{\Delta \rhd E_1 : \mathsf{b}_1 \quad \Delta \rhd E_2 : \mathsf{b}_2}{\Delta \rhd E_1 \otimes E_2 : \mathsf{b}}(\otimes : \mathsf{b}_1 \times \mathsf{b}_2 \to \mathsf{b})$$

$$[\mathit{let}]\,\frac{\Delta, x{:}\,\sigma_1 \rhd E_2 : \sigma_2 \quad \Delta \rhd E_1 : \sigma_1}{\Delta \rhd \mathbf{let}\ x \Leftarrow E_1\ \mathbf{in}\ E_2 : \sigma_2}$$

Equational Rules $\boxed{\mathsf{vPCF} \vdash \Delta \rhd E_1 = E_2 : \sigma}$

The equational rules distinguish a subset of (possibly non-closed) terms, called *values*, ranged over by meta-variable V.
Values $V ::= \ell \mid x \mid \lambda x.E \mid d$

Let x, f, and g range over variables in the rules.

Congruence rules: $=$ is a congruence. (Detailed rules omitted)

Equations for let-expressions

[*unit*]	$\Delta \rhd \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ x = E : \sigma$
[*assoc*]	$\Delta \rhd \mathbf{let}\ x_2 \Leftarrow (\mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E$ $= \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ E : \sigma$
[*let*.β]	$\Delta \rhd \mathbf{let}\ x \Leftarrow V\ \mathbf{in}\ E = E\{V/x\} : \sigma$
[*let.app*]	$\Delta \rhd E_1\,E_2 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ x_1\,x_2 : \sigma$
[*let.fix*]	$\Delta \rhd \mathbf{fix}\,E = \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ \mathbf{fix}\,x : \sigma_1 \to \sigma_2$
[*let.if*]	$\Delta \rhd \mathbf{if}\,E_1\,E_2\,E_3 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{if}\,x\,E_2\,E_3 : \sigma$
[*let*.$\otimes$]	$\Delta \rhd E_1 \otimes E_2 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ x_1 \otimes x_2 : \mathsf{b}$

Other rules

[β_v]	$\Delta \rhd (\lambda x.E)V = E\{V/x\} : \sigma$
[η_v]	$\Delta \rhd (\lambda x.f\,x) = f : \sigma_1 \to \sigma_2$
[*fix-dinat*]	$\Delta \rhd \mathbf{fix}\,(f \circ g) = f(\mathbf{fix}\,(g \circ f)) : \sigma_1 \to \sigma_2$
[*if*-tt]	$\Delta \rhd \mathbf{if}\,\mathsf{tt}\,E_2\,E_3 = E_2 : \sigma$
[*if*-ff]	$\Delta \rhd \mathbf{if}\,\mathsf{ff}\,E_2\,E_3 = E_3 : \sigma$
[*if*-η]	$\Delta \rhd \mathbf{if}\,x\,E\,E = E : \sigma$
[$\otimes$]	$\Delta \rhd \ell_1 \otimes \ell_2 = \ell : \mathsf{b} \quad (\ell_1 \otimes \ell_2 = \ell)$

Figure 14: One-level call-by-value language vPCF

C.3 Annotation erasure

First, we display the equational rules of vPCF in Figure 14. As mentioned in Section 4, vPCF is an instance of Moggi's computational λ-calculus [35].

Lemma C.14 (Erasure of θ-typed values). *Let V be a* vPCF^2*-value (see Figure 9). If $\Gamma \rhd V : \theta$ (i.e., V is of a substitution-safe type), then $|V|$ is a* vPCF*-value.*

Lemma 4.5 (Annotation erasure). *If* $\mathsf{vPCF}^2 \vdash \textcircled{v}\Delta \rhd [B]E : \tau$ *and* $\mathsf{vPCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$*, then* $\mathsf{vPCF} \vdash \Delta \rhd \mathbf{let}^*\ |B|\ \mathbf{in}\ |E| = \mathbf{let}^*\ |B'|\ \mathbf{in}\ |V| : |\tau|$.

Proof. We prove by induction on $\Delta \rhd [B]E \Downarrow [B']V$ that

$$\Delta; B \rhd |E| = \mathbf{let}^*\ |B' - B|\ \mathbf{in}\ V : |\tau|,$$

from which the conclusion follows using the congruence rule for **let**. We write $\delta_{i,j}B$ $(j \geq i)$ for $B_j - B_i$.

Case [*lit*], [*lam*]**:** There is nothing to prove.

Case [*app*]**:** Applying Theorem 4.2 to $E_2 \Downarrow V'$, and then using Lemma C.14, we have that $|V'|$ is a vPCF-value. Now we have

$$\begin{aligned}
&|E_1\, E_2| \equiv |E_1|\, |E_2| \\
\stackrel{i.h.}{=}\ &(\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ |\lambda x.E'|)\, (\mathbf{let}^*\ |\delta_{2,3}B|\ \mathbf{in}\ |V'|) \\
=\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ \mathbf{let}^*\ |\delta_{2,3}B|\ \mathbf{in}\ (\lambda x.|E'|)\, |V'| \\
\stackrel{[\beta_v]}{=}\ &\mathbf{let}^*\ |\delta_{1,3}B|\ \mathbf{in}\ |E'|\{V'/x\} \\
\stackrel{i.h.}{=}\ &\mathbf{let}^*\ |\delta_{1,3}B|\ \mathbf{in}\ \mathbf{let}^*\ |\delta_{3,4}B|\ \mathbf{in}\ V \equiv \mathbf{let}^*\ |\delta_{1,4}B|\ \mathbf{in}\ V.
\end{aligned}$$

Case [*if*-tt]**:**
$$\begin{aligned}
&|\mathbf{if}\, E_1\, E_2\, E_3| \equiv \mathbf{if}\, |E_1|\, |E_2|\, |E_3| \\
\stackrel{i.h.}{=}\ &\mathbf{if}\, (\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ \mathrm{tt})\, |E_2|\, |E_3| \\
=\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ (\mathbf{if}\, \mathrm{tt}\, |E_2|\, |E_3|) = \mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ |E_2| \\
\stackrel{i.h.}{=}\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ \mathbf{let}^*\ |\delta_{2,3}B|\ \mathbf{in}\ |V| \equiv \mathbf{let}^*\ |\delta_{1,3}B|\ \mathbf{in}\ |V|.
\end{aligned}$$

Case [*fix*]**:**
$$\begin{aligned}
&|\mathbf{fix}\, E| \equiv \mathbf{fix}\, |E| \stackrel{i.h.}{=} \mathbf{fix}\, (\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ |\lambda x.E'|) \\
=\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ \mathbf{fix}\, (\lambda x.|E'|) \\
=\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ |E'|\{\mathbf{fix}\, (\lambda x.|E'|)/x\} \\
\stackrel{i.h.}{=}\ &\mathbf{let}^*\ |\delta_{1,2}B|\ \mathbf{in}\ \mathbf{let}^*\ |\delta_{2,3}B|\ \mathbf{in}\ |V| \equiv \mathbf{let}^*\ |\delta_{1,3}B|\ \mathbf{in}\ |V|.
\end{aligned}$$

Case [*if*-ff],[$\otimes$]**:** Simple; similar to the case of Rule ([*if*-tt]).

Case [$\underline{\mathit{lift}}$],[$\underline{\mathit{var}}$],[$\underline{\mathit{cst}}$]**:** Trivial.

Case [$\underline{\mathit{lam}}$]: $\Delta; B \rhd |\underline{\lambda} x.E| \equiv \lambda x.|E| \sim_\alpha \lambda y.|E\{y/x\}|$

$$\stackrel{i.h.}{=} \lambda y.(\mathbf{let}^* \ |B'| \ \mathbf{in} \ |\mathcal{O}|)$$
$$\equiv \mathbf{let}^* \ |[B - B]| \ \mathbf{in} \ |\lambda y.(\mathbf{let}^* \ B' \ \mathbf{in} \ \mathcal{O})|.$$

For this step we use the induction hypothesis; we also apply the congruence for λ-abstractions.

Case [$\underline{\mathit{app}}$]: $|E_1 \underline{@} E_2| \equiv |E_1|\,|E_2|$

$$\stackrel{i.h.}{=} (\mathbf{let}^* \ |\delta_{1,2}B| \ \mathbf{in} \ |\mathcal{O}_1|)\,(\mathbf{let}^* \ |\delta_{2,3}B| \ \mathbf{in} \ |\mathcal{O}_2|)$$
$$= \mathbf{let}^* \ |\delta_{1,2}B| \ \mathbf{in} \ \mathbf{let}^* \ |\delta_{2,3}B| \ \mathbf{in} \ |\mathcal{O}_1|\,|\mathcal{O}_2|$$
$$\equiv \mathbf{let}^* \ |\delta_{1,3}B| \ \mathbf{in} \ |\mathcal{O}_1 \underline{@} \mathcal{O}_2|.$$

Case [$\underline{\mathit{let}}$]: $|\underline{\mathbf{let}} \ x \Leftarrow E_1 \ \underline{\mathbf{in}} \ E_2| \equiv \mathbf{let} \ x \Leftarrow |E_1| \ \mathbf{in} \ |E_2|$

$$\sim_\alpha \mathbf{let} \ y \Leftarrow |E_1| \ \mathbf{in} \ |E_2\{y/x\}|$$
$$\stackrel{i.h.}{=} \mathbf{let} \ y \Leftarrow (\mathbf{let}^* \ |B' - B| \ \mathbf{in} \ |\mathcal{O}_1|) \ \mathbf{in} \ (\mathbf{let}^* \ |B''| \ \mathbf{in} \ |\mathcal{O}_2|)$$
$$= \mathbf{let}^* \ |B' - B| \ \mathbf{in} \ \mathbf{let} \ y \Leftarrow |\mathcal{O}_1| \ \mathbf{in} \ (\mathbf{let}^* \ |B''| \ \mathbf{in} \ |\mathcal{O}_2|).$$

Case [#]: $|\#E| = |E| \stackrel{i.h.}{=} \mathbf{let}^* \ |B' - B| \ \mathbf{in} \ |\mathcal{O}|$ □

$$= \mathbf{let}^* \ |B' - B| \ \mathbf{in} \ \mathbf{let} \ x \Leftarrow |\mathcal{O}| \ \mathbf{in} \ x$$
$$= \mathbf{let}^* \ |[B', x : \sigma = \mathcal{O}] - B| \ \mathbf{in} \ |x|.$$

Lemma 4.5 has the following immediate corollary for complete programs.

Theorem 4.6 (Annotation erasure for complete programs). *If* $\mathsf{vPCF}^2 \vdash\ \rhd E : ⓔ\sigma$ *and* $\mathsf{vPCF}^2 \vdash E \searrow \mathcal{O}$, *then* $\mathsf{vPCF} \vdash\ \rhd |E| = |\mathcal{O}| : \sigma$.

Proof. Assume that $\rhd E : ⓔ\sigma$ and $E \searrow \mathcal{O}$. That is, $\rhd [\cdot] E \Downarrow [B]\mathcal{O}'$ where $\mathcal{O} \equiv \mathbf{let}^* \ B \ \mathbf{in} \ \mathcal{O}'$). By Lemma 4.5, we have $\rhd |E| = \mathbf{let}^* \ |B| \ \mathbf{in} \ |\mathcal{O}'| \equiv |\mathcal{O}| : \sigma$. □

C.4 Native implementation

C.4.1 A more "realistic" language: $\langle\mathsf{v}\rangle\mathsf{PCF}^2$

The semantics of vPCF^2 could re-evaluate values of code types—though such re-evaluation does not change the result. Consider, for example, the evaluation of the term $(\lambda x.x)\,(\underline{\mathsf{print}}\underline{@}(\$_{\mathsf{int}}1+2))$. The semantics first evaluates the argument to the code value $\underline{\mathsf{print}}\underline{@}(\$_{\mathsf{int}}3)$, then proceeds to evaluate $x\{\underline{\mathsf{print}}\underline{@}(\$_{\mathsf{int}}3)/x\} \equiv \underline{\mathsf{print}}\underline{@}(\$_{\mathsf{int}}3)$, which needs a complete recursive descent though it is already a value. Such re-evaluation does not change the semantics, but it nevertheless is less efficient, and does not model the actual implementation.

To establish a native implementation, we thus consider a variant of vPCF^2, $\langle\mathsf{v}\rangle\mathsf{PCF}^2$, which marks evaluated terms with angle brackets and prevents them from re-evaluation.[12] The detailed changes of $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ over vPCF^2 are given in

[12]The reader might notice that angle brackets here have a similar functionality to *quote* in Lisp and Scheme. But they serve two different purposes: angle brackets here prevent re-evaluation at the semantics level, thereby removing an artifact of the substitution-based semantics and bringing the semantics closer to the actual implementation; quote in Lisp

Syntax

Raw terms $E ::= \ldots \mid \langle \mathcal{O} \rangle$

Values $V ::= \ell \mid \lambda x.E \mid \langle \mathcal{O} \rangle$

Typing Judgment Add the following rule

(Dynamic) $$[\mathit{eval'd}]\ \frac{\Gamma \rhd \mathcal{O} : \tau}{\Gamma \rhd \langle \mathcal{O} \rangle : \tau}$$

Evaluation Semantics The dynamic part is replaced by the following rules.

$$[\mathit{eval'd}]\ \frac{}{\langle \mathcal{O} \rangle \Downarrow \langle \mathcal{O} \rangle} \quad [\underline{\mathit{lift}}]\ \frac{E \Downarrow \ell}{\$_{\mathsf{b}} E \Downarrow \langle \$_{\mathsf{b}} \ell \rangle} \quad [\underline{\mathit{var}}]\ \frac{}{x \Downarrow \langle x \rangle} \quad [\underline{\mathit{cst}}]\ \frac{}{\underline{d} \Downarrow \langle \underline{d} \rangle}$$

$$[\underline{\mathit{lam}}]\ \frac{\Delta, y : \sigma; B \rhd [\cdot] E\{y/x\} \Downarrow [B'] \langle \mathcal{O} \rangle \quad y \notin \mathit{dom}\, B \cup \mathit{dom}\, \Delta}{\Delta \rhd [B] \underline{\lambda} x.E \Downarrow [B] \langle \underline{\lambda} y.\underline{\mathbf{let}}^*\ B'\ \underline{\mathbf{in}}\ \mathcal{O} \rangle}$$

$$[\underline{\mathit{app}}]\ \frac{E_1 \Downarrow \langle \mathcal{O}_1 \rangle \quad E_2 \Downarrow \langle \mathcal{O}_2 \rangle}{E_1 \underline{@} E_2 \Downarrow \langle \mathcal{O}_1 \underline{@} \mathcal{O}_2 \rangle}$$

$$[\underline{\mathit{let}}]\ \frac{\begin{array}{c}\Delta \rhd [B] E_1 \Downarrow [B'] \langle \mathcal{O}_1 \rangle \quad \Delta, y : \sigma; B \rhd [\cdot] E_2\{y/x\} \Downarrow [B''] \langle \mathcal{O}_2 \rangle \\ y \notin \mathit{dom}\, B' \cup \mathit{dom}\, \Delta\end{array}}{\Delta \rhd [B] \underline{\mathbf{let}}\ y \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 \Downarrow [B'] \langle \underline{\mathbf{let}}\ x \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ (\underline{\mathbf{let}}^*\ B''\ \underline{\mathbf{in}}\ \mathcal{O}_2) \rangle}$$

$$[\#]\ \frac{\Delta \rhd [B] E \Downarrow [B'] \langle \mathcal{O} \rangle \quad x \notin \mathit{dom}\, B' \cup \mathit{dom}\, \Delta}{\Delta \rhd [B] \# E \Downarrow [B', x : \sigma = \mathcal{O}] \langle x \rangle}$$

Figure 15: Changes of ⟨v⟩PCF² over vPCF²

Figure 15. Note that binders consist of only already evaluated terms, so there is no need to mark them with angle brackets.

The two languages vPCF² and ⟨v⟩PCF² are effectively equivalent, through the following operation to remove the angle brackets in ⟨v⟩PCF²-terms:

Definition C.15 (Unbracketing). *The unbracketing of an ⟨v⟩PCF²-term E, noted as $\mathit{unbr}(E)$, is the vPCF²-term resulted from E by removing all the angle brackets in it, i.e., $\mathit{unbr}(\langle E \rangle) \equiv E$ and all other constructs are translated homomorphically.*
We have that ⟨v⟩PCF² $\vdash \Gamma \rhd E : \tau$ implies ⟨v⟩PCF² $\vdash \Gamma \rhd \mathit{unbr}(E) : \tau$.

Theorem C.16 (Equivalence of vPCF² and ⟨v⟩PCF²).

1. *If vPCF² $\vdash \Delta \rhd [B] \mathit{unbr}(E) \Downarrow [B'] V'$, then there exists a ⟨v⟩PCF²-value V such that ⟨v⟩PCF² $\vdash \Delta \rhd [B] E \Downarrow [B'] V$ and $\mathit{unbr}(V) \equiv V'$.*

allows programmers to distinguish data from the surrounding programs. A similar concern arises when implementing, e.g., syntactic theories.

2. *If* $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash \Delta \triangleright [B]E \Downarrow [B']V$*, then it also holds that* $\mathsf{vPCF}^2 \vdash \Delta \triangleright [B]\mathit{unbr}(E) \Downarrow [B']\mathit{unbr}(V)$*.*

Proof. Simple induction. □

Corollary C.17. *If* $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash \Delta \triangleright E : ⓔ\sigma$*, and* $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash \Delta \triangleright [B]E \Downarrow [B']V$*, then* V *is of the form* $\langle\mathcal{O}\rangle$*.*

Proof. By Theorem C.16, we have $\mathsf{vPCF}^2 \vdash \Delta \triangleright [B]\mathit{unbr}(E) \Downarrow [B']\mathit{unbr}(V)$. Using the type preservation of vPCF^2, we have $\mathit{unbr}(V)$ is of the form $\mathcal{O}$. Since V is a $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-value, it must be $\langle\mathcal{O}\rangle$. □

Note that every vPCF^2-term E is also a $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-term, and $\mathit{unbr}(E) \equiv E$. Thus, to evaluate a closed vPCF^2-term E of type ⓔσ (i.e., $\mathsf{vPCF}^2 \vdash \triangleright E : ⓔ\sigma$), we can evaluate E using $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ semantics. Either the evaluation does not terminate, which implies, by Theorem C.16(1), that the evaluation E in vPCF^2 semantics does not terminate. Or $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash E \searrow \langle\mathcal{O}\rangle$ (defined appropriately using Corollary C.17), which, by Theorem C.16(2), implies that $\mathsf{vPCF}^2 \vdash E \searrow \mathcal{O}$.

Now that we have established the equivalence of vPCF^2 and $\langle\mathsf{v}\rangle\mathsf{PCF}^2$, it suffices to give a native implementation of $\langle\mathsf{v}\rangle\mathsf{PCF}^2$. In the rest of the section, we will only work with $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ and leave it (instead of vPCF^2) implicit whenever possible.

C.4.2 The implementation language: $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$

Here we present the implementation language $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$, which can be viewed as a subset of Standard ML [33]. In detail, $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$ is vPCF with dynamic constants removed, and enriched with a global state and an inductive type Λ for representing $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ code types.

$$\begin{array}{rcl}\Lambda & = & \mathsf{VAR}\ \mathit{of}\ \mathsf{int} \mid \mathsf{LIT_b}\ \mathit{of}\ \mathsf{b} \mid \mathsf{CST}\ \mathit{of}\ \mathsf{const} \mid \mathsf{LAM}\ \mathit{of}\ \mathsf{int} \times \Lambda \\ & \mid & \mathsf{APP}\ \mathit{of}\ \Lambda \times \Lambda \mid \mathsf{LET}\ \mathit{of}\ \mathsf{int} \times \Lambda \times \Lambda\end{array}$$

For the state, we only model the part in which we are interested. A state S thus consists of a cell n for keeping the counter of name generation, and a cell B for keeping the accumulated bindings: $S = \langle n, B\rangle$. We also use three specialized primitives (they can be implemented using SML operations := (set) and ! (get), as shown in Figure 16): $\mathsf{genvar}()$ generates a new variable name using the counter; $\mathsf{addBind}(E : \Lambda) : \Lambda$ adds a binding $x = E$ to the accumulated bindings such that x is a newly generated name, and returns the variable x; finally $\mathsf{letBind}(E : \Lambda)$ creates a dynamic local scope for accumulating bindings during the evaluation of term E, and inserts the accumulated bindings after the evaluation of E. The three primitives are governed by the following typing rules. The evaluation semantics of $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$ is given in Figure 17.

$$\frac{}{\Gamma \triangleright \mathsf{genvar}() : \mathsf{int}} \qquad \frac{\Gamma \triangleright E : \Lambda}{\Gamma \triangleright \mathsf{addBind}(E) : \Lambda} \qquad \frac{\Gamma \triangleright E : \Lambda}{\Gamma \triangleright \mathsf{letBind}(E) : \Lambda}$$

```
val n : int ref              = ref 0        (* name generation counter *)
val B : (int * exp) list ref = ref []         (* accumulated bindings *)

fun init () = n := 0                                (* reset the counter *)
fun genvar () =                                           (* genvar() *)
    !n  before n := ! n + 1
fun letBind e_thunk =             (* letBind(E) ≡letBind (fn () => E) *)
    let val b = ! B  before B := []
        val r = e_thunk ()
        fun genLet [] body =
            body
          | genLet ((x, e) :: rest) body =
            genLet rest (LET(x, e, body))
    in
        genLet (! B) r  before B := b
    end
fun addBind e =                          (* addBind(E) ≡addBind(E) *)
    let val name = genvar () in
        (B := (name, e) :: ! B); (VAR name)
    end
```

Figure 16: ML implementation of $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$-primitives

C.4.3 Native embedding

Definition C.18 (Embedding translation $\{\!|-|\!\}_{\mathsf{v}\epsilon}$ of $\langle\mathsf{v}\rangle\mathsf{PCF}^2$ into $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$). $\boxed{\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash \Gamma \rhd E : \tau} \Longrightarrow \boxed{\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|\Gamma|\!\}_{\mathsf{v}\epsilon} \rhd \{\!|E|\!\}_{\mathsf{v}\epsilon} : \{\!|\tau|\!\}_{\mathsf{v}\epsilon}}$

Types : $\{\!|Ⓔ\sigma|\!\}_{\mathsf{v}\epsilon} = \{\!|Ⓥ\sigma|\!\}_{\mathsf{v}\epsilon} = \Lambda$, $\{\!|\mathsf{b}|\!\}_{\mathsf{v}\epsilon} = \mathsf{b}$, $\{\!|\tau_1 \to \tau_2|\!\}_{\mathsf{v}\epsilon} = \{\!|\tau_1|\!\}_{\mathsf{v}\epsilon} \to \{\!|\tau_2|\!\}_{\mathsf{v}\epsilon}$

Terms : $\{\!|\$_{\mathsf{b}}E|\!\}_{\mathsf{v}\epsilon} = \mathsf{LIT}_{\mathsf{b}}(\{\!|E|\!\}_{\mathsf{v}\epsilon})$, $\{\!|\underline{d}|\!\}_{\mathsf{v}\epsilon} = \mathsf{CST}(\wr d \int)$, $\{\!|\underline{\lambda} x.E|\!\}_{\mathsf{v}\epsilon} = \underline{\lambda}^{\mathsf{v}\epsilon}\lambda x.\{\!|E|\!\}_{\mathsf{v}\epsilon}$,
$\{\!|E_1 \underline{@} E_2|\!\}_{\mathsf{v}\epsilon} = \mathsf{APP}(\{\!|E_1|\!\}_{\mathsf{v}\epsilon}, \{\!|E_2|\!\}_{\mathsf{v}\epsilon})$, $\{\!|\#E|\!\}_{\mathsf{v}\epsilon} = \underline{\#}^{\mathsf{v}\epsilon}(\{\!|E|\!\}_{\mathsf{v}\epsilon})$,
$\{\!|\underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2|\!\}_{\mathsf{v}\epsilon} = \underline{\mathsf{let}}^{\mathsf{v}\epsilon}\{\!|E_1|\!\}_{\mathsf{v}\epsilon}\lambda x.\{\!|E_2|\!\}_{\mathsf{v}\epsilon}$, $\{\!|\langle\mathcal{O}\rangle|\!\}_{\mathsf{v}\epsilon} = \{\!|\mathcal{O}|\!\}_{\langle\rangle\epsilon}$

"Evaluated" terms:
$\{\!|\$_{\mathsf{b}}\ell|\!\}_{\langle\rangle\epsilon} = \mathsf{LIT}_{\mathsf{b}}(\ell)$, $\{\!|v_i|\!\}_{\langle\rangle\epsilon} = \mathsf{VAR}(i)$, $\{\!|\underline{\lambda} v_i.\mathcal{O}|\!\}_{\langle\rangle\epsilon} = \mathsf{LAM}(i, \{\!|\mathcal{O}|\!\}_{\langle\rangle\epsilon})$,
$\{\!|\mathcal{O}_1 \underline{@} \mathcal{O}_2|\!\}_{\langle\rangle\epsilon} = \mathsf{APP}(\{\!|\mathcal{O}_1|\!\}_{\langle\rangle\epsilon}, \{\!|\mathcal{O}_2|\!\}_{\langle\rangle\epsilon})$, $\{\!|\underline{d}|\!\}_{\langle\rangle\epsilon} = \mathsf{CST}(\wr d \int)$,
$\{\!|\underline{\mathbf{let}}\ v_i \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ \mathcal{O}_2|\!\}_{\langle\rangle\epsilon} = \mathsf{LET}(i, \{\!|\mathcal{O}_1|\!\}_{\langle\rangle\epsilon}, \{\!|\mathcal{O}_2|\!\}_{\langle\rangle\epsilon})$

Bindings:
$\{\!|\cdot|\!\}_{\mathsf{v}\epsilon} = \mathsf{nil}$
$\{\!|B, v_i : \sigma = \mathcal{O}|\!\}_{\mathsf{v}\epsilon} = (i, \{\!|\mathcal{O}|\!\}_{\langle\rangle\epsilon}) :: \{\!|B|\!\}_{\mathsf{v}\epsilon}$

where we use the following terms:

$$
\begin{array}{rcl}
\underline{\lambda}^{\mathsf{v}\epsilon} & \equiv & \lambda f.\mathbf{let}\ i \Leftarrow \mathsf{genvar}()\ \mathbf{in}\ \mathsf{LAM}(i, \mathsf{letBind}(f\,(\mathsf{VAR}(i)))) \\
\underline{\mathsf{let}}^{\mathsf{v}\epsilon} & \equiv & \lambda e.\lambda f.\mathbf{let}\ i \Leftarrow \mathsf{genvar}()\ \mathbf{in}\ \mathsf{LET}(i, e, \mathsf{letBind}(f\,(\mathsf{VAR}(i)))) \\
\underline{\#}^{\mathsf{v}\epsilon} & \equiv & \lambda e.\mathsf{addBind}(e)
\end{array}
$$

Note that the embedding translation is partial: among "evaluated" terms

State $S ::= \langle n, B\rangle \quad (n \in \mathbb{N})$
Bindings $B ::= \mathsf{nil} \mid (n, V) :: B \quad (n \in \mathbb{N})$

Judgment Form $\boxed{\mathsf{vPCF}^{\wedge,\mathsf{st}} \vdash E, S \Downarrow V, S'}$

We use the following abbreviations.

$$\frac{E_1 \Downarrow V_1 \quad \ldots \quad E_n \Downarrow V_n}{E \Downarrow V} \equiv \frac{E_1, S_1 \Downarrow V_1, S_2 \quad \ldots \quad E_n, S_n \Downarrow V_n, S_{n+1}}{E, S_1 \Downarrow V, S_{n+1}}$$

$$\underline{\mathsf{LET}}^* \ (n_1, V_1) :: \ldots :: (n_m, V_m) :: \mathsf{nil}\ \underline{\mathbf{in}}\ V \equiv \mathsf{LET}(n_m, V_m, \ldots \mathsf{LET}(n_1, V_1, V) \ldots)$$

(Core)

$$[\mathit{app}]\ \frac{E_1 \Downarrow \lambda x.E' \quad E_2 \Downarrow V' \quad E'\{V'/x\} \Downarrow V}{E_1\, E_2 \Downarrow V} \qquad [\mathit{if}\text{-}\mathsf{tt}]\ \frac{E_1 \Downarrow \mathsf{tt} \quad E_2 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V}$$

$$[\mathit{if}\text{-}\mathsf{ff}]\ \frac{E_1 \Downarrow \mathsf{ff} \quad E_3 \Downarrow V}{\mathbf{if}\, E_1\, E_2\, E_3 \Downarrow V} \qquad [\mathit{fix}]\ \frac{E \Downarrow \lambda x.E' \quad E'\{\mathbf{fix}\,(\lambda x.E')/x\} \Downarrow V}{\mathbf{fix}\, E \Downarrow V}$$

$$[\otimes]\ \frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{E_1 \otimes E_2 \Downarrow V} (V_1 \otimes V_2 = V)$$

(Term and State)

$$[\mathit{cons}]\ \frac{E_1 \Downarrow V_1 \quad \ldots \quad E_n \Downarrow V_n}{c(E_1, \ldots, E_n) \Downarrow c(V_1, \ldots, V_n)} (c \in \{\mathsf{VAR}, \mathsf{LIT}_\mathsf{b}, \mathsf{CST}, \mathsf{LAM}, \mathsf{APP}, \mathsf{LET}\})$$

$$[\mathit{genvar}]\ \frac{}{\mathsf{genvar}(), \langle n, B\rangle \Downarrow n, \langle n+1, B\rangle}$$

$$[\mathit{addBind}]\ \frac{E, S \Downarrow V, \langle n, B\rangle}{\mathsf{addBind}(E), S \Downarrow \mathsf{VAR}(n), \langle n+1, (n, V) :: B\rangle}$$

$$[\mathit{letbind}]\ \frac{E, \langle n, \mathsf{nil}\rangle \Downarrow V, \langle n', B'\rangle}{\mathsf{letBind}(E), \langle n, B\rangle \Downarrow \underline{\mathsf{LET}}^*\ B'\ \underline{\mathbf{in}}\ V, \langle n', B\rangle}$$

Figure 17: The evaluation semantics of $\mathsf{vPCF}^{\wedge,\mathsf{st}}$

(i.e., those enclosed in angle brackets), only those whose variables (both free and bound) range in the set $\{v_i : i \in \mathbf{Z}\}$ have a translation. A $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-term is $\mathsf{v}\epsilon$-*embeddable* if all its "evaluated" subterms satisfy this condition. Clearly, all vPCF^2-terms, viewed as $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-terms, are without "evaluated" subterms and thus $\mathsf{v}\epsilon$-embeddable. In the following, when we write $\{\!|E|\!\}_{\mathsf{v}\epsilon}$, we implicitly state that E is $\mathsf{v}\epsilon$-embeddable.

Note also that α-conversion is not preserved by the embedding translation: bound variables v_i are translated to integer i under the translation for the "evaluated" terms. As a consequence, a general substitution lemma of the following form fails for this translation.

$$\{\!|E\{E'/x\}|\!\}_{\mathsf{v}\epsilon} \sim_\alpha \{\!|E|\!\}_{\mathsf{v}\epsilon}\{\{\!|E'|\!\}_{\mathsf{v}\epsilon}/x\}$$

The problem occurs when the capture-free substitution goes under a dynamic λ-abstraction *inside* an "evaluated" term, which possibly requires the renaming of the bound variable. For example, the substitution $\langle\underline{\lambda} v_1.x \underline{@} v_1\rangle\{\langle v_1\rangle/x\}$ should rename the bound variable v_1 and yield $\langle\underline{\lambda} v_2.v_1 \underline{@} v_2\rangle$: the substitution of the translated terms, $\{\!|\langle\underline{\lambda} v_1.x \underline{@} v_1\rangle|\!\}_{\mathsf{v}\epsilon}\{\{\!|\langle v_1\rangle|\!\}_{\mathsf{v}\epsilon}/x\} \equiv \mathsf{LAM}(1, \mathsf{APP}(\mathsf{VAR}(1), \mathsf{VAR}(1)))$, is different from the translation of the substitution $\mathsf{LAM}(2, \mathsf{APP}(\mathsf{VAR}(1), \mathsf{VAR}(2)))$.

Fortunately, such problematic substitutions do not actually occur in an evaluation. Intuitively, all variables occurring inside an evaluated term are already generated, and thus not amenable to substitution. This intuition is captured by the following well-formedness condition.

Definition C.19 (Well-formed $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-terms). *A $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-term E is* well-formed*, if for all its "evaluated" subterms $\langle\mathcal{O}\rangle$, no free variable in $\mathcal{O}$ is bound by a static or dynamic λ-abstraction.*

The class of well-formed terms is closed under evaluation. It includes all the vPCF^2-terms.

Lemma C.20. *If $\lambda x.E$ and E' are both well-formed, then so is $E\{E'/x\}$.*

Lemma C.21. *If E is well formed and $[B]E \Downarrow [B']E'$, then all terms occurring in its derivation, which include E', are well formed.*

Proof. By induction on the derivation. We use the previous lemma for function application. □

For well-formed terms, capture-free substitution used in the β-reduction has a substitution lemma.

Lemma C.22 (Substitution lemma for well-formed terms). *If $\lambda x.E$ is well-formed, then $\{\!|E\{E'/x\}|\!\}_{\mathsf{v}\epsilon} \equiv \{\!|E|\!\}_{\mathsf{v}\epsilon}\{\{\!|E'|\!\}_{\mathsf{v}\epsilon}/x\}$.*[13]

[13]Here, strict syntactic equality is possible, because we are free to pick a representative from the α-equivalent class formed by the possible results of the substitution.

Proof. By the definition of a well-form term, x does not appear inside any "evaluated" subterms of E. Now we proceed by induction. In particular, for the case of $\langle\mathcal{O}\rangle$, we have $\langle\mathcal{O}\rangle\{E'/x\} \equiv \langle\mathcal{O}\rangle$; and for the case of $\lambda y.E'$, renaming y does not affect variable names that appear inside the evaluated terms, again by the definition of a well-formed term. $\square$

The correctness proof of the native embedding also uses a few other notations.

- We write $\mathcal{V}_I$ for the set of variable names indexed by set I, where I is a subset of the integer set $\mathbf{Z}$, i.e., $\mathcal{V}_I = \{v_i : i \in I\}$.
- We write $[n]$ for the set $\{0, 1, \ldots, n-1\}$.
- For $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash ⓥ\Delta \rhd E : \tau$ where $dom\,\Delta \subseteq \mathcal{V}_{\mathbf{Z}}$, we write $\{\!|E|\!\}^C_{\mathsf{v}\epsilon}$ ("closed embedding translation") for the term $\{\!|E|\!\}_{\mathsf{v}\epsilon}\{\Phi^C\}$ where the substitution $\Phi^C = \{\mathsf{VAR}(i)/v_i : i \in \mathbf{Z}\}$. It is clear that the resulting term must be closed.

We are ready to prove the correctness of the native embedding. We first prove the soundness of the implementation (that its evaluation gives a correct result whenever it terminates), and then prove the completeness of the implementation (that its evaluation must terminate if the source program terminates according to the $\langle\mathsf{v}\rangle\mathsf{PCF}^2$-semantics). As for the notation, we write iV for the **i**implementation of a value V.

Lemma C.23 (Soundness of the Implementation). *If*

(a) $ⓥ\Delta \rhd [B]E : \tau$ *where* $dom\,\Delta \cup dom\,B \subseteq \mathcal{V}_{[n]}$,

(b) E *is well-formed, and*

(c) $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|E|\!\}^C_{\mathsf{v}\epsilon}, S \Downarrow iV, S'$ *where* $S = \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle$,

then there exist a $\langle\mathsf{v}\rangle\mathsf{PCF}^2$*-value* V*, a* $\langle\mathsf{v}\rangle\mathsf{PCF}^2$*-binding* B'*, and an integer* $n' \geq n$ *such that*

- $\langle\mathsf{v}\rangle\mathsf{PCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$,
- $S' = \langle n', \{\!|B'|\!\}^C_{\mathsf{v}\epsilon}\rangle$, $iV = \{\!|V|\!\}^C_{\mathsf{v}\epsilon}$, *and*
- $dom\,\Delta \cup dom\,B' \subseteq \mathcal{V}_{[n']}$.

Proof. By induction on the derivation of $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|E|\!\}^C_{\mathsf{v}\epsilon}, S \Downarrow iV, S'$ (Condition (c)). It is a routine task to ensure Condition (a) during the induction. To ensure Condition (b) during the induction, we use Lemma C.21.

We perform a case analysis on E.

Case $E \equiv \ell$: Simple.

Case $E \equiv \langle \mathcal{O} \rangle$**:** Since $\{\!\!\{\langle \mathcal{O} \rangle\}\!\!\}^C_{\mathsf{v}\epsilon}$ must be a value already (a simple inductive proof), we have, by inversion, that $iV \equiv \{\!\!\{\langle \mathcal{O} \rangle\}\!\!\}^C_{\mathsf{v}\epsilon}$ and $S' = S$. Therefore, we put $B' = B$, $V = \langle \mathcal{O} \rangle$ and $n' = n$: $\langle \mathcal{O} \rangle \Downarrow \langle \mathcal{O} \rangle$.

Case $E \equiv v_i$**:** Since $\{\!\!\{v_i\}\!\!\}^C_{\mathsf{v}\epsilon} \equiv \mathsf{VAR}(i)$ is a value, we have that $iV \equiv \mathsf{VAR}(i)$ and $S' = S$. We can put $B' = B$, $V = \langle v_i \rangle$, and $n' = n$: $v_i \Downarrow \langle v_i \rangle$.

Case $E \equiv \lambda x.E'$**:** Since $\{\!\!\{\lambda x.E'\}\!\!\}^C_{\mathsf{v}\epsilon} \equiv \lambda x.\{\!\!\{E'\}\!\!\}^C_{\mathsf{v}\epsilon}$ is a value, we have that $iV \equiv \lambda x.\{\!\!\{E'\}\!\!\}^C_{\mathsf{v}\epsilon}$ and $S' = S$. We can put $B' = B$, $V = \lambda x.E'$, and $n' = n$: $\lambda x.E' \Downarrow \lambda x.E'$.

Case $E \equiv E_1\, E_2$**:** We have that $\{\!\!\{E_1\, E_2\}\!\!\}^C_{\mathsf{v}\epsilon} \equiv \{\!\!\{E_1\}\!\!\}^C_{\mathsf{v}\epsilon}\, \{\!\!\{E_2\}\!\!\}^C_{\mathsf{v}\epsilon}$. By inversion, the last step in the derivation of $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!\!\{E_1\}\!\!\}^C_{\mathsf{v}\epsilon}\, \{\!\!\{E_2\}\!\!\}^C_{\mathsf{v}\epsilon}, S \Downarrow iV, S'$ must be of the following form (let $S_1 = S$ and $S_4 = S'$).

$$\frac{\{\!\!\{E_1\}\!\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow \lambda x.iE', S_2 \quad \{\!\!\{E_2\}\!\!\}^C_{\mathsf{v}\epsilon}, S_2 \Downarrow iV', S_3 \quad iE'\{iV'/x\}, S_3 \Downarrow iV, S_4}{\{\!\!\{E_1\}\!\!\}^C_{\mathsf{v}\epsilon}\, \{\!\!\{E_2\}\!\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow iV, S_4}$$

where $S_1 = \langle n, \{\!\!\{B\}\!\!\}^C_{\mathsf{v}\epsilon} \rangle$. We have the following reasoning:

(1) By induction hypothesis 1 (and by the fact that only λ-abstractions translates to λ-abstractions under the $\mathsf{v}\epsilon$-translation), there exist B_2, E', and n_2 such that

  a. $\Delta \rhd [B]E_1 \Downarrow [B_2]\lambda x.E'$, $\{\!\!\{E'\}\!\!\}^C_{\mathsf{v}\epsilon} = iE'$; and

  b. $n_2 \geq n$, $S_2 = \langle n_2, \{\!\!\{B_2\}\!\!\}^C_{\mathsf{v}\epsilon} \rangle$, $\mathit{dom}\,\Delta \cup \mathit{dom}\,B_2 \subseteq \mathcal{V}_{[n_2]}$.

(2) By induction hypothesis 2 and (1.b), there exist B_3, V', and n_3 such that

  a. $\Delta \rhd [B_2]E_2 \Downarrow [B_3]V'$, $\{\!\!\{V'\}\!\!\}^C_{\mathsf{v}\epsilon} = iV'$; and

  b. $n_3 \geq n_2$, $S_3 = \langle n_3, \{\!\!\{B_3\}\!\!\}^C_{\mathsf{v}\epsilon} \rangle$, $\mathit{dom}\,\Delta \cup \mathit{dom}\,B_3 \subseteq \mathcal{V}_{[n_3]}$.

(3) By Lemma C.21, $\lambda x.E'$ is well-formed, thus by Lemma C.22, we have

$$\begin{array}{rcl} iE'\{iV'/x\} & \equiv & \{\!\!\{E'\}\!\!\}^C_{\mathsf{v}\epsilon}\{\{\!\!\{V'\}\!\!\}^C_{\mathsf{v}\epsilon}/x\} \\ & \equiv & \{\!\!\{E'\{V'/x\}\}\!\!\}^C_{\mathsf{v}\epsilon} \end{array}$$

(4) By (2), (3), and induction hypothesis 3, there exist B_4, V, and n_4 such that

  a. $\Delta \rhd [B_3]E'\{V'/x\} \Downarrow [B_4]V$, $\{\!\!\{V\}\!\!\}^C_{\mathsf{v}\epsilon} = iV$; and

  b. $n_4 \geq n_3$, $S_4 = \langle n_4, \{\!\!\{B_4\}\!\!\}^C_{\mathsf{v}\epsilon} \rangle$, $\mathit{dom}\,\Delta \cup \mathit{dom}\,B_4 \subseteq \mathcal{V}_{[n_4]}$.

Finally, one application of the $\langle \mathsf{v} \rangle\mathsf{PCF}^2$ evaluation rule ([*app*]) to (1.a), (2.a), and (4.a) yields: $\Delta \rhd [B_1]E_1\, E_2 \Downarrow [B_4]V$.

Case $E \equiv \mathbf{fix}\, E_1$: We have that $\{\!|\mathbf{fix}\, E_1|\!\}^C_{\mathsf{v}\epsilon} \equiv \mathbf{fix}\, \{\!|E_1|\!\}^C_{\mathsf{v}\epsilon}$. By inversion, the derivation ends with

$$\frac{\{\!|E_1|\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow \lambda x.iE', S_2 \quad iE'\{\mathbf{fix}\, \lambda x.iE'/x\}, S_2 \Downarrow iV, S_3}{\mathbf{fix}\, \{\!|E_1|\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow iV, S_3}$$

where $S_1 = \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle$. Then we reason as follows:

(1) By induction hypothesis 1 (and by the fact that only λ-abstractions translate to λ-abstractions under the $\mathsf{v}\epsilon$-translation), there exist B_2, E', and n_2 such that

  a. $\Delta \rhd [B]E_1 \Downarrow [B_2]\lambda x.E'$, $\{\!|E'|\!\}^C_{\mathsf{v}\epsilon} = iE'$; and
  b. $n_2 \geq n$, $S_2 = \langle n_2, \{\!|B_2|\!\}^C_{\mathsf{v}\epsilon}\rangle$, $\mathit{dom}\,\Delta \cup \mathit{dom}\, B_2 \subseteq \mathcal{V}_{[n_2]}$.

(2) Applying Lemma C.21 to (1.a) gives that $\lambda x.E'$ is well-formed, and thus by Lemma C.22, we have

$$\begin{array}{rcl} iE'\{\mathbf{fix}\, \lambda x.iE'/x\} & \equiv & \{\!|E'|\!\}^C_{\mathsf{v}\epsilon}\{\{\!|\mathbf{fix}\, \lambda x.E'|\!\}^C_{\mathsf{v}\epsilon}/x\} \\ & \equiv & \{\!|E'\{\mathbf{fix}\, \lambda x.E'/x\}|\!\}^C_{\mathsf{v}\epsilon} \end{array}$$

(3) By (2) and induction hypothesis 2, there exist B_3, V, and n_3 such that

  a. $\Delta \rhd [B_2]E'\{\mathbf{fix}\, \lambda x.E'/x\} \Downarrow [B_3]V$, $\{\!|V|\!\}^C_{\mathsf{v}\epsilon} = iV$; and
  b. $n_3 \geq n_2$, $S_3 = \langle n_3, \{\!|B_3|\!\}^C_{\mathsf{v}\epsilon}\rangle$, $\mathit{dom}\,\Delta \cup \mathit{dom}\, B_3 \subseteq \mathcal{V}_{[n_3]}$.

Finally, one application of the evaluation rule ([*fix*]) to (1.a) and (3.a) yields: $\Delta \rhd [B_1]\mathbf{fix}\, E_1 \Downarrow [B_3]V$.

Case $E \equiv \mathbf{if}\, E_1\, E_2\, E_3$ or $E \equiv E_1 \otimes E_2$: Similar to the proofs for $E_1\, E_2$ and for $\mathbf{fix}\, E_1$, and only simpler, since these cases are free of the complication introduced by capture-free substitution.

Case $E \equiv \$_{\mathsf{b}} E_1$, $E \equiv E_1 \underline{@} E_2$, or $E \equiv \underline{d}$: Simple.

Case $E \equiv \underline{\lambda} x.E_1$: We have $\{\!|\underline{\lambda} x.E_1|\!\}^C_{\mathsf{v}\epsilon} \equiv \underline{\lambda}^{\mathsf{v}\epsilon}\, ((\lambda x.\{\!|E_1|\!\}_{\mathsf{v}\epsilon})\{\Phi^C\})$. By a few inversions from $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|\underline{\lambda} x.E_1|\!\}^C_{\mathsf{v}\epsilon}, \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle \Downarrow iV, S'$, we have the following two immediate subderivations (omitting some trivial branches)

$$\frac{}{\mathsf{genvar}(), \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle \Downarrow n, \langle n+1, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle}$$

$$\dfrac{\dfrac{\dfrac{\{\!|E_1|\!\}_{\mathsf{v}\epsilon}\{\mathsf{VAR}(n)/x\}\{\Phi^C\}, \langle n+1, \mathsf{nil}\rangle \Downarrow iV', \langle n', iB'\rangle}{((\lambda x.\{\!|E_1|\!\}_{\mathsf{v}\epsilon})\{\Phi^C\})\,(\mathsf{VAR}(n)), \langle n+1, \mathsf{nil}\rangle \Downarrow iV', \langle n', iB'\rangle}}{L, \langle n+1, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle \Downarrow \underline{\mathsf{LET}}^*\ iB'\ \underline{\mathbf{in}}\ iV', \langle n', \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle}}{\mathsf{LAM}(n, L), \langle n+1, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle \Downarrow \mathsf{LAM}(n, \underline{\mathsf{LET}}^*\ iB'\ \underline{\mathbf{in}}\ iV'), \langle n', \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle}$$

where $L \equiv \mathsf{letBind}(((\lambda x.\{\!|E_1|\!\}_{\mathsf{v}\epsilon})\{\Phi^C\})\,(\mathsf{VAR}(n)))$.

Note that

$$\{\!|E_1|\!\}_{\mathsf{v}\epsilon}\{\mathsf{VAR}(n)/x\}\{\Phi^C\} \equiv \{\!|E_1\{v_n/x\}|\!\}_{\mathsf{v}\epsilon}\{\mathsf{VAR}(n)/v_n\}\{\Phi^C\} \equiv \{\!|E_1\{v_n/x\}|\!\}_{\mathsf{v}\epsilon}\{\Phi^C\}$$

(the second equality follows from the definition of Φ^C). From $ⓥ\Delta \rhd [B]\underline{\lambda} x.E_1 : ⓔ(\sigma_1 \to \sigma_2)$ and $\mathit{dom}\,\Delta \cup \mathit{dom}\,B \subseteq \mathcal{V}_{[n]}$, it follows that $ⓥ\{\Delta, v_n : \sigma_1; B\} \rhd E_1\{v_n/x\} : ⓥ\sigma_2$ and $\mathit{dom}(\Delta, v_n : \sigma_1) \cup \mathit{dom}\,B \subseteq \mathcal{V}_{[n+1]}$. Furthermore, $E_1\{v_n/x\}$ is clearly well-formed. Thus, by induction hypothesis, there exist B'' and V'' such that

a. $\Delta, v_n : \sigma_1; B \rhd [\cdot]E_1\{v_n/x\} \Downarrow [B']V''$ and $\{\!|V''|\!\}^C_{\mathsf{v}\epsilon} = iV'$. Type Preservation shows that $V'' \equiv \langle \mathcal{O} \rangle$ for some $\mathcal{O}$; and

b. $iB' = \{\!|B'|\!\}^C_{\mathsf{v}\epsilon}$, $\mathit{dom}\,\Delta \cup \mathit{dom}\,B' \subseteq \mathcal{V}_{[n']}$, and $n' \geq n + 1$.

Finally, put $V = \langle \underline{\lambda} v_n.\underline{\mathbf{let}}^*\ B''\ \underline{\mathbf{in}}\ \mathcal{O} \rangle$ and $B' = B$, and apply the evaluation rule ([*lam*]) to (a). Noting that $v_n \notin \mathcal{V}_{[n]} \supseteq \mathit{dom}\,\Delta \cup \mathit{dom}\,B$, we have that $\Delta \rhd [B]E_1 : [B]V$. It is easy to check that $\{\!|V|\!\}_{\mathsf{v}\epsilon} \equiv \mathsf{LAM}(n, \underline{\mathsf{LET}}^*\ iB'\ \underline{\mathbf{in}}\ iV')$.

Case $E \equiv \underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2$: Similar to the case of $\underline{\lambda} x.E_1$.

Case $E \equiv \# E_1$: By inversion on the assumption, the derivation ends with

$$\frac{\{\!|E_1|\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow iV', S_2 \quad \mathsf{addBind}(iV'), S_2 \Downarrow iV, S'}{\underline{\#}^{\mathsf{v}\epsilon}\ \{\!|E_1|\!\}^C_{\mathsf{v}\epsilon}, S_1 \Downarrow iV, S'}$$

where $S_1 = \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon} \rangle$. Then we reason as follows.

(1) By induction hypothesis 1, there exist B_2, V', and n_2 such that

   a. $\Delta \rhd [B]E_1 \Downarrow [B_2]V'$ and $\{\!|V'|\!\}^C_{\mathsf{v}\epsilon} = iV'$. By type preservation, $V' \equiv \langle \mathcal{O} \rangle$ for some $\mathcal{O}$; and

   b. $n_2 \geq n$, $S_2 = \langle n_2, \{\!|B_2|\!\}^C_{\mathsf{v}\epsilon} \rangle$, and $\mathit{dom}\,\Delta \cup \mathit{dom}\,B_2 \subseteq \mathcal{V}_{[n_2]}$.

(2) Inverting the second premise, we have that $iV = \mathsf{VAR}(n_2)$, and $S' = \langle n_2 + 1, (n_2, iV') :: \{\!|B_2|\!\}^C_{\mathsf{v}\epsilon} \rangle$. We can put $V \equiv \langle v_{n_2} \rangle$, $B' \equiv (B_2, v_{n_2+1} : \sigma = \mathcal{O})$, and $n' \equiv n_2 + 1$; it is easy to check that $\{\!|V|\!\}^C_{\mathsf{v}\epsilon} = iV$ and $\langle n', \{\!|B'|\!\}^C_{\mathsf{v}\epsilon} \rangle = S'$. Note also that $v_{n_2} \notin \mathcal{V}_{[n_2]} \supseteq \mathit{dom}\,\Delta \cup \mathit{dom}\,B_2$. Now we can apply the rule ([#]) to get the result.

□

Lemma C.24 (Completeness of the Implementation). *If*

(a) $ⓥ\Delta \rhd [B]E : \tau$ *where* $\mathit{dom}\,\Delta \cup \mathit{dom}\,B \subseteq \mathcal{V}_{[n]}$,

(b) E *is well-formed, and*

(c) $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B_2]V$,

then there exist B_2', V', and $n' \geq n$ such that

- $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B_2']V'$,
- $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|E|\!\}^C_{\mathsf{v}\epsilon}, \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon} \rangle \Downarrow V', \langle n', \{\!|B_2'|\!\}^C_{\mathsf{v}\epsilon} \rangle$, *and*
- $dom\, \Delta \cup dom\, B' \subseteq \mathcal{V}_{[n']}$.

Proof. By induction on the height of the derivation of $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B_2]V$ (Condition (c)). It is a routine task to ensure Condition (a) during the induction. To ensure Condition (b) during the induction, we use Lemma C.21.

We perform case analysis on the last rule used in the derivation.

Case [*lit*], [*lam*]**:** Simple.

Case [*app*]**:** We combine the induction hypotheses and Lemma C.22.

Case [*fix*]**:** We combine the induction hypotheses and Lemma C.22.

Case [*if*-tt], [*if*-ff], [$\otimes$]**:** We combine the induction hypotheses, and use the fact that constants of base types translate to themselves.

Case [*eval'd*], [$\underline{lift}$], [$\underline{var}$], [$\underline{cst}$], [$\underline{app}$]**:** Simple. See also the corresponding cases in the soundness proof (Lemma C.23).

Case [$\underline{lam}$]**:** The derivation tree takes the following form

$$\frac{\overset{\mathcal{D}}{\Delta, y : \sigma; B \rhd [\cdot]E\{y/x\} \Downarrow [B']\langle \mathcal{O} \rangle} \quad y \notin dom\, B \cup dom\, \Delta}{\Delta \rhd [B]\underline{\lambda} x.E \Downarrow [B]\langle \underline{\lambda} y.\underline{\mathbf{let}}^*\ B'\ \underline{\mathbf{in}}\ \mathcal{O} \rangle}$$

Since $v_n \notin dom\, \Delta \cup dom\, B$, we have

$$(\Delta, y : \sigma; B \rhd E\{y/x\}) \sim_\alpha (\Delta, v_n : \sigma; B \rhd E\{v_n/x\})\,.$$

By Theorem C.13, there exist B'' and $\mathcal{O}'$ such that $(\Delta, v_n : \sigma; B \rhd [B']\mathcal{O}) \sim_\alpha (\Delta', v_n : \sigma; B \rhd [B'']\mathcal{O}')$ and there is a a derivation for

$$\Delta, v_n : \sigma; B \rhd [\cdot]E\{v_n/x\} \Downarrow [B'']\langle \mathcal{O}' \rangle$$

that has the same size as derivation $\mathcal{D}$. Noting further that $E\{v_n/x\}$ is well-formed, we can apply the induction hypothesis to conclude that $\exists B''', \mathcal{O}'', n' \geq n+1$ such that the following hold.

(1) $\Delta, v_n : \sigma; B \rhd [\cdot]E\{v_n/x\} \Downarrow [B''']\langle \mathcal{O}'' \rangle$. Again, by Theorem C.13, we have that $(\Delta, v_n : \sigma; B \rhd [B'']\mathcal{O}') \sim_\alpha (\Delta, v_n : \sigma; B \rhd [B''']\mathcal{O}'')$.

(2) $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|E\{v_n/x\}|\!\}^C_{\mathsf{v}\epsilon}, \langle n+1, \mathsf{nil} \rangle \Downarrow \{\!|\langle \mathcal{O}'' \rangle|\!\}^C_{\mathsf{v}\epsilon}, \langle n', \{\!|B'''|\!\}^C_{\mathsf{v}\epsilon} \rangle$.

(3) $dom\,\Delta \cup \{v_n\} \cup dom\,B' \subseteq \mathcal{V}_{[n']}$.

We can then construct derivations for

- $\langle \mathsf{v} \rangle\mathsf{PCF}^2 \vdash \Delta \rhd [B]E[B]\langle \underline{\mathbf{let}}^* \; B''' \; \underline{\mathbf{in}} \; \mathcal{O}''\rangle \Downarrow$, by applying rule ([$\underline{lam}$]) to (1), and
- $\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \{\!|\underline{\lambda}x.E|\!\}^C_{\mathsf{v}\epsilon}, \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle \Downarrow \{\!|\langle \underline{\lambda} v_n.\underline{\mathbf{let}}^* \; B''' \; \underline{\mathbf{in}} \; \mathcal{O}''\rangle|\!\}^C_{\mathsf{v}\epsilon}, \langle n, \{\!|B|\!\}^C_{\mathsf{v}\epsilon}\rangle$, using a derivation in the form appeared in the case $E \equiv \underline{\lambda}x.E_1$ of the soundness proof (Lemma C.23)

The conclusion follows immediately.

Case [$\underline{let}$]**:** Similar to the case of rule ([$\underline{lam}$]).

Case [#]**:** We build the derivation from the induction hypothesis. See also the corresponding cases in the soundness proof (Lemma C.23). □

Definition C.25 (Simulating the evaluation in $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$). *Let* $\mathsf{vPCF}^2 \vdash \rhd E : \bigcirc\sigma$. *We write* $simEval(E,t)$ *for a* $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$*-term* $t : \Lambda$, *if* $\exists S'.\mathsf{vPCF}^{\Lambda,\mathsf{st}} \vdash \mathsf{letBind}(\{\!|E|\!\}_{\mathsf{v}\epsilon}), \langle 0, \mathsf{nil}\rangle \Downarrow t, S'$.

Theorem C.26 (Total correctness). *Let* $\mathsf{vPCF}^2 \vdash \rhd E : \bigcirc\sigma$.

1. *If* $\mathsf{vPCF}^2 \vdash E \searrow \mathcal{O}$ *for some* $\mathcal{O}$*, then there is a term* $\mathcal{O}'$ *such that* $\mathcal{O}' \sim_\alpha \mathcal{O}$ *and* $simEval(E, \{\!|\mathcal{O}'|\!\}_{\mathsf{v}\epsilon})$.
2. *If* $simEval(E,t)$ *for some* $t : \Lambda$*, then there is a term* $\mathcal{O}$ *such that* $t \equiv \{\!|\mathcal{O}|\!\}_{\mathsf{v}\epsilon}$ *and* $\mathsf{vPCF}^2 \vdash E \searrow \mathcal{O}$.

Proof. We combine Theorem C.16, Lemma C.23, and Lemma C.24. □

C.5 Call-by-value type-directed partial evaluation

C.5.1 Semantic correctness

Lemma C.27. *For all types* σ, $\mathsf{nPCF} \vdash \; \rhd |{\downarrow^\sigma}| = \lambda x.x : \sigma \to \sigma$ *and* $\mathsf{nPCF} \vdash \rhd |{\uparrow_\sigma}| = \lambda x.x : \sigma \to \sigma$.

Proof. By a straightforward induction on type τ. □

Theorem C.28 (Semantic correctness of TDPE). *If* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$ *and* $\mathsf{vPCF}^2 \vdash NF(E) \Downarrow \mathcal{O}$*, then* $\mathsf{vPCF} \vdash \; \rhd |\mathcal{O}| = |E| : \sigma$.

(Note that the two erasures are different: One operates on vPCF^2-terms, the other on $\mathsf{vPCF}^{\mathsf{tdpe}}$-terms.)

Proof. Similar to the proof of the corresponding theorem in the call-by-name case, Theorem 2.12, but using Lemma C.27 and Theorem 4.6 instead. □

C.5.2 Syntactic correctness

Theorem 4.8 (Refined type preservation). *If* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B]E : \tau$ *and* $\mathsf{vPCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$*, then* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B']V : \tau$*.*

Proof. (Sketch) Similar to the proof of Theorem 4.2. As always, the most nontrivial case is the rule ([*app*]), for which we prove a substitution lemma for the refined type system (similar to Lemma B.9.) ☐

Corollary 4.9 (Refined type preservation for complete programs). *If* $\blacktriangleright E : \textcircled{e}^{nc}(\sigma)$ *and* $E \searrow \mathcal{O}$*, then* $\blacktriangleright \mathcal{O} : \textcircled{e}^{nc}(\sigma)$*.*

Theorem 4.10 (Normal-form code types). *If* V *is an* vPCF^2*-value (Figure 8), and* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright V : \textcircled{v}^{X}(\sigma)$ *where* X *is av, nv, bd, or nc, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^X |\mathcal{O}| : \sigma$*.*

Proof. Similar to the proof of Theorem 2.14. ☐

Lemma 4.11. *(1) The extraction functions (Figure 10c) have the following normal-form types (writing* $\sigma^{\bigcirc\mathsf{nv}}$ *for* $\sigma\{\textcircled{v}^{nv}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*.)*

$$\blacktriangleright \downarrow^{\sigma} : \sigma^{\bigcirc\mathsf{nv}} \to \textcircled{v}^{nv}(\sigma), \blacktriangleright \uparrow_{\sigma} : \textcircled{v}^{av}(\sigma) \to \sigma^{\bigcirc\mathsf{nv}}.$$

(2) If $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi$*, then* $\mathsf{vPCF}^2 \vdash \{\!|\Gamma|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\varphi|\!\}^{\mathsf{nv}}_{\mathsf{ri}}$*, where* $\{\!|\varphi|\!\}^{\mathsf{nv}}_{\mathsf{ri}} = \varphi\{\textcircled{v}^{nv}(\mathsf{b})/\mathsf{b}^{\mathfrak{d}} : \mathsf{b} \in \mathbb{B}\}$*.*

Proof.
(1) By induction on type σ.

Case $\sigma = \mathsf{b}$: Because at the base type, $\mathsf{b}^{\bigcirc\mathsf{nv}} = \textcircled{v}^{nv}(\mathsf{b})$, we just need to show: $\blacktriangleright \lambda x.x : \textcircled{v}^{nv}(\mathsf{b}) \to \textcircled{v}^{nv}(\mathsf{b})$ for $\downarrow^{\mathsf{b}}$, and $\blacktriangleright \lambda x.x : \textcircled{v}^{av}(\mathsf{b}) \to \textcircled{v}^{nv}(\mathsf{b})$ for $\uparrow_{\mathsf{b}}$. This is simple.

Case $\sigma = \sigma_1 \to \sigma_2$: Noting that $(\sigma_1 \to \sigma_2)^{\bigcirc\mathsf{nv}} = \sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}$, we give the following typing derivations, in the compact style used in the proof of Lemma 2.15.

- For $\blacktriangleright \downarrow^{\sigma_1 \to \sigma_2} : (\sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}) \to \textcircled{v}^{nv}(\sigma_1 \to \sigma_2)$:

$$f : \sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}, x : \textcircled{v}^{var}(\sigma_1) \blacktriangleright \underbrace{\downarrow^{\sigma_2}(\overbrace{f\,(\underbrace{\uparrow_{\sigma_1} x}_{\sigma_1^{\bigcirc\mathsf{nv}}})}^{\sigma_2^{\bigcirc\mathsf{nv}}})}_{\textcircled{v}^{nv}(\sigma_2)} : \textcircled{e}^{nc}(\sigma_2)$$

Note the use of implicit coercions for term x and for $\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x))$.

- For $\blacktriangleright \uparrow_{\sigma_1 \to \sigma_2} : \text{ⓥ}^{av}(\sigma_1 \to \sigma_2) \to (\sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}})$:

$$e : \text{ⓥ}^{av}(\sigma_1 \to \sigma_2), x : \sigma_1^{\bigcirc\mathsf{nv}} \blacktriangleright \uparrow_{\sigma_2}(\underbrace{\#(\overbrace{e\underline{@}(\underbrace{\downarrow^{\sigma_1} x}_{\text{ⓥ}^{nv}(\sigma_1)})}^{\text{ⓔ}^{bd}(\sigma_2)})}_{\text{ⓥ}^{var}(\sigma_2)}) : \sigma_2^{\bigcirc\mathsf{nv}}$$

(2) By a simple induction on $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \Gamma \rhd E : \varphi$. For the case where $E \equiv d^{\mathfrak{d}}$ with $Sg(d) = \sigma$, we use the typing of $\uparrow_\sigma$ from part (1) and the fact that $\{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \equiv \{\!|\sigma\{\mathsf{b}^{\mathfrak{d}}/\mathsf{b} : b \in \mathbb{B}\}|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \equiv \sigma\{\text{ⓥ}^{nv}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\} \equiv \sigma^{\bigcirc\mathsf{nv}}$. □

Theorem 4.12. *If* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$*, then* $\mathsf{vPCF}^2 \vdash\!\blacktriangleright NF(E) : \text{ⓥ}^{nv}(\sigma)$*.*

Proof. By Lemma 4.11(2), we have $\mathsf{vPCF}^2 \vdash\!\blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nv}}_{\mathsf{ri}}$. Since $\{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \equiv \sigma^{\bigcirc\mathsf{nv}}$, applying $\downarrow^\sigma : \sigma^{\bigcirc\mathsf{nv}} \to \text{ⓥ}^{nv}(\sigma)$ (Lemma 4.11(1)) to $\{\!|E|\!\}_{\mathsf{ri}}$ yields the conclusion. □

Corollary C.29 (Syntactic correctness of TDPE). *For* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$*, if* $\mathsf{vPCF}^2 \vdash NF(E) \searrow V$*, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\mathsf{vPCF} \vdash \Delta \rhd^{nc} |\mathcal{O}| : \sigma$*.*

Proof. We use Theorem 4.12, Corollary 4.9, and Theorem 4.10. □

D Notation and symbols

Meta-variables and fonts

E	(one-level or two-level) terms	
x, y, z	variables	
τ (resp. σ)	two-level (resp. one-level) types	8,9, 22,53
φ	two-level types in TDPE languages	16,25
θ	substitution-safe types (vPCF^2)	22
Γ (resp. Δ)	two-level/one-level typing contexts	8,9
B	accumulated bindings	23
b	base types	7
ℓ	literals (constants of base types)	7
d	dynamic constants in signature Sg	8
V	values (canonical terms)	8,23
$\mathcal{O}$	code-typed values	8,23
S	state	59
Sans serif ($\mathsf{bool}, \mathsf{CST}$)	syntax	
Underlined ($\underline{\lambda} x., \underline{@}$)	dynamic constructs	8,22
$d^{\mathfrak{d}}, \mathsf{b}^{\mathfrak{d}}$	dynamic constants and base types in TDPE languages	16,25

Language and Judgments

$\mathsf{L} \vdash J$	judgment J holds in language L.	

General judgments

$\Gamma \triangleright E : \tau$	term-in-context: "term E is of type τ under context Γ"	
$\Gamma \triangleright E_1 = E_2 : \tau$	equation-in-context: "E_1 and E_2 are equal terms of type τ under context Γ"	

nPCF^2: call-by-name two-level language 8

$E \Downarrow V$	evaluation of a statically closed term	8
$\Gamma \blacktriangleright E : \tau$	term-in-context with refined typing for $\beta\eta$-normal object terms	18

nPCF: call-by-name one-level language 9

$\Delta \triangleright^X E : \sigma$ ($X \in \{at, nf\}$)	typing judgments for $\beta\eta$-normal forms	18

nPCF^Λ: CBN language with a term type 12

$\mathsf{nPCF}^{\mathsf{tdpe}}$ and $\mathsf{vPCF}^{\mathsf{tdpe}}$: two-level languages for TDPE 16,25

$\mathsf{v}\Lambda$ and $\mathsf{n}\Lambda$: pure simply typed λ-calculus 11,34

vPCF^{2} and $\langle\mathsf{v}\rangle\mathsf{PCF}^{2}$: call-by-value two-level languages 23,56

$\Gamma \rhd [B]E : \tau$	binder-term-in-context	24,46
$\Gamma \rhd [B]$	binder-in-context	46
$\Gamma \rhd [B] \geq [B']$	binder extension	46
$B - B'$	difference of binder B and its prefix B'	46
$(\Gamma \rhd E) \sim_\alpha (\Gamma' \rhd E')$	α-equivalence for terms-in-context	51
$(\Gamma \rhd [B]E) \sim_\alpha (\Gamma' \rhd [B']E')$	α-equivalence for binder-terms-in-context	51
$\Gamma \rhd [B]E \Downarrow [B']V$	evaluation of a binder-term-in-context	23
$E \searrow \mathcal{O}$	evaluation of a complete program	24
$\Gamma \blacktriangleright E : \tau$	term-in-context with refined typing for λ_c-normal object terms	27

vPCF: call-by-value one-language with effects 53

$\Delta \rhd^X E : \sigma$ $X = av, nv, bd, nc$	typing judgments for λ_c-normal forms	27

$\mathsf{vPCF}^{\Lambda,\mathsf{st}}$: CBV language with a term type and state 59

$E, S \Downarrow V, S'$	evaluation	59

General notations

$[\![-]\!]$	(denotational) meaning function	13,35
$\{\!\vert-\vert\!\}$	syntactic translation	
$\{\!\vert-\vert\!\}_{\mathsf{n}\epsilon}$	native embedding of nPCF^{2} into nPCF^{Λ}	13
$\{\!\vert-\vert\!\}_{\mathsf{v}\epsilon}, \{\!\vert-\vert\!\}_{\langle\rangle\epsilon}$	native embedding of $\langle\mathsf{v}\rangle\mathsf{PCF}^{2}$ into $\mathsf{vPCF}^{\Lambda,\mathsf{st}}$	58
$\{\!\vert-\vert\!\}_{\mathsf{p}\kappa}$	Plotkin's CPS transformations	11,34
$\{\!\vert-\vert\!\}_{\mathsf{df}^2\kappa}, \{\!\vert-\vert\!\}_{\mathsf{df}\kappa}$	Danvy and Filinski's one-pass CPS transformation	11,34
$\vert - \vert$	annotation erasure of two-level terms	10,16,24
$\equiv$	strict syntactic equality	7
$\sim_\alpha$	α-equivalence	7
$\bigcirc\sigma, ⓥ\sigma, \bigcirc^{at}(\sigma), \ldots$	code types	8,18, 23,27
$\mathcal{D}(-)$	decoding of a term representation	13
$unbr(-)$	unbracketing of $\langle\mathsf{v}\rangle\mathsf{PCF}^{2}$-terms	56
$E\{\theta\}$	application of the substitution θ to E	

TDPE-specific notations

$\downarrow^\sigma$	reification function at type σ	16,25
$\uparrow_\sigma$	reflection function at type σ	16,25
$\{\!\vert-\vert\!\}_{\mathsf{ri}}$	residualizing instantiation	16,25
$NF(-)$	static normalization function	16,25

References

[1] Alan Bawden. Quasiquotation in Lisp. In Olivier Danvy, editor, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 4–12, San Antonio, Texas, January 1999. ACM Press. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[4] Cristiano Calcago, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *27th International Colloquium on Automata, Languages, and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, Switzerland, July 2000.

[5] Olivier Danvy. Type-directed partial evaluation. In Steele [48], pages 242–257.

[6] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School, LNCS 1706*, pages 367–411, Copenhagen, Denmark, July 1998.

[7] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[8] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[9] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in Haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 343–358, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-34.

[10] Rowan Davies. A temporal-logic approach to binding-time analysis. In Edmund M. Clarke, editor, *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, pages 184–195, New Brunswick, New Jersey, July 1996.

[11] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [48], pages 258–283. Extended version to appear in *Journal of the ACM*, and also available as CMU Technical Report number CMU-CS-99-153, August 1999.

[12] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming, LNCS 1702*, pages 378–395, Paris, France, September 1999.

[13] Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer-Verlag.

[14] Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In John Mitchell, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science*, pages 193–202, Trento, Italy, July 1999.

[15] Yoshihiko Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[16] Carsten K. Gomard. A self-applicable partial evaluator for the lambda-calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, 1992.

[17] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. In Lawall [30], pages 22–32. Extended version to appear in the journal *Higher-Order and Symbolic Computation*, 14(2/3), 2001.

[18] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, Massachusetts, 1992.

[19] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 1994.

[20] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[21] John McCarthy *et al.* *LISP 1.5 Programmer's Manual*. The MIT Press, Cambridge, Massachusetts, 1962.

[22] Neil D. Jones, editor. *Special issue on Partial Evaluation*, Journal of Functional Programming, Vol. 3, Part 3. Cambridge University Press, July 1993.

[23] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[24] Neil D. Jones and Steven S. Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Languages*, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[25] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[26] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

[27] Eugene E. Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In William L. Scherlis and John H. Williams, editors, *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, Cambridge, Massachusetts, August 1986.

[28] Eugene E. Kohlbecker and Mitchell Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In Michael J. O'Donnell, editor, *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 77–84, München, West Germany, January 1987.

[29] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[30] Julia L. Lawall, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, SIGPLAN Notices, Vol. 34, No 11, Boston, Massachusetts, November 2000. ACM Press.

[31] Peter Lee. *Realistic Compiler Generation*. The MIT Press, 1989.

[32] Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings, LNCS 193*, pages 219–224, Brooklyn, New York, June 1985.

[33] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[34] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[35] Eugenio Moggi. Computational lambda-calculus and monads. In Rohit Parikh, editor, *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989.

[36] Eugenio Moggi. Functor categories and two-level languages. In Maurice Nivat, editor, *Foundations of Software Science and Computation Structure, First International Conference, LNCS 1378*, pages 211–225, Lisbon, Portugal, 1998.

[37] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An Idealized MetaML: Simpler, and more expressive. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming, LNCS 1576*, pages 193–207, Amsterdam, Netherlands, March 1999.

[38] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

[39] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[40] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[41] Flemming Nielson and Hanne Riis Nielson. Multi-level lambda-calculi: an algebraic description. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, pages 338–354, Dagstuhl, Germany, February 1996.

[42] Flemming Nielson and Hanne Riis Nielson. Prescriptive frameworks for multi-level lambda-calculi. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 193–202, Amsterdam, The Netherlands, June 1997.

[43] Jens Palsberg. Correctness of binding-time analysis. In Jones [22], pages 347–363.

[44] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988.

[45] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[46] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[47] Gordon D. Plotkin. An illative theory of relations. In Richard Cooper, K. Mukai, and John Perry, editors, *Situation Theory and Its Applications (Vol. 1)*, pages 133–146. CSLI, Stanford, California, 1990.

[48] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[49] Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approah. In Lawall [30], pages 12–21. Extended version titled "A hybrid approach to online and offline partial evaluation" to appear in the journal *Higher-Order and Symbolic Computation*, 14(2/3), 2001.

[50] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival (extended abstract). In Lawall [30], pages 34–43.

[51] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[52] Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones [22], pages 365–387.

[53] Philip Wickline, Peter Lee, and Frank Pfenning. Run-time code generation and Modal-ML. In Keith D. Cooper, editor, *Proceedings of the ACM SIGPLAN'98 Conference on Programming Languages Design and Implementation*, pages 224–235, Montréal, Canada, June 1998.

[54] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.

Recent BRICS Report Series Publications

RS-00-46 **Zhe Yang.** ***Reasoning About Code-Generation in Two-Level Languages*. December 2000. 74 pp.**

RS-00-45 **Ivan B. Damgård and Mads J. Jurik.** ***A Generalisation, a Simplification and some Applications of Paillier's Probabilistic Public-Key System*. December 2000. 18 pp. Appears in Kim, editor, *Fourth International Workshop on Practice and Theory in Public Key Cryptography*, PKC '01 Proceedings, LNCS 1992, 2001, pages 119–136. This revised and extended report supersedes the earlier BRICS report RS-00-5.**

RS-00-44 **Bernd Grobauer and Zhe Yang.** ***The Second Futamura Projection for Type-Directed Partial Evaluation*. December 2000. Appears in *Higher-Order and Symbolic Computation* 14(2–3):173–219 (2001). This revised and extended report supersedes the earlier BRICS report RS-99-40 which in turn was an extended version of Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00 Proceedings, 2000, pages 22–32.**

RS-00-43 **Claus Brabrand, Anders Møller, Mikkel Ricky Christensen, and Michael I. Schwartzbach.** ***PowerForms: Declarative Client-Side Form Field Validation*. December 2000. 21 pp. Appears in *World Wide Web Journal*, 4(3), 2000.**

RS-00-42 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***The `<bigwig>` Project*. December 2000. 25 pp. To appear in ACM Transactions on Internet Technology, 2002.**

RS-00-41 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***The DSD Schema Language and its Applications*. December 2000. 32 pp. Appears in *Automated Software Engineering*, 9(3):285–319,2002. Shorter version appears in Heimdahl, editor, *3rd ACM SIGSOFT Workshop on on Formal Methods in Software Practice*, FMSP '00 Proceedings, 2000 , pages 101–111.**