

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.

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.

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

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

$$E = \text{VAR of string} \mid \text{LAM of string} * E \mid \text{APP of } E * E$$

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

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 $eta \triangleq \lambda f.\underline{\lambda}x.f @ x$, 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 $\underline{\lambda}x.\underline{\lambda}x.x @ x$, 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 $\underline{\lambda}x.(\lambda f.\underline{\lambda}x.f @ x)x$ does not evaluate to $\underline{\lambda}x.\underline{\lambda}x.x @ x$, but to $\underline{\lambda}x.\lambda y.x @ y$. This precise issue is referred to as “hygienic macro expansion” in Kohlbecker’s work [25, 26].

*Extended Abstract of a technical report [52].

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

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. [21, 22, 23]: 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 $\lambda x.(\lambda f.\lambda x.f @ x)x$, 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, 33] (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 [34]. 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 [42]. 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-name “PCF of two-level languages”, following Moggi [33]). 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-value 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, our general framework, in particular the desired properties and the corresponding proof obligations they support. With this framework in mind, in Section 4 we briefly examine the technical considerations in designing $vPCF^2$, and leave the detailed development to Appendix A: There, we present the language, 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 part in Section 6. The detailed proofs and development can be found in the full version of this paper [52].

Notational conventions: Because we consider several different languages, we write $L \vdash J$ to assert a judgment J

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

in the language \mathbb{L} , or we write simply J when \mathbb{L} is clear from the context. We write \equiv for strict syntactic equality, and \sim_α for equality up to α -conversion. Operations (syntactic translations) defined on types τ , say $\{\tau\}$, are homomorphically extended to apply to contexts: $\{x_1 : \tau_1, \dots, x_n : \tau_n\} \equiv x_1 : \{\tau_1\}, \dots, x_n : \{\tau_n\}$. A type-preserving translation $\{_ \}$ of terms-in-contexts in language \mathbb{L}_1 into ones in language \mathbb{L}_2 is declared in the form $\boxed{\mathbb{L}_1 \vdash \Delta \triangleright E : \sigma} \implies \boxed{\mathbb{L}_2 \vdash \{\Delta\} \triangleright \{E\} : \{\sigma\}}$. 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 nPCF²

We present a canonical call-by-name (CBN) two-level language nPCF² (Section 2.1), cast the example of a one-pass CPS transformation as an nPCF² program (Section 2.2), and use an erasure argument to prove its correctness (Section 2.3). Building on a native embedding of nPCF² into a conventional language (Section 2.4), we formulate CBN TDPE in nPCF² and show its semantic correctness as well as its syntactic correctness (Sections 2.5 and 2.6).

2.1 Syntax and semantics

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

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 $(\Gamma \triangleright E : \tau)$ and the evaluation semantics $(E \Downarrow V)$ of nPCF² 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 $\text{if}_\sigma : \text{bool} \rightarrow \sigma \rightarrow \sigma$ in Sg .

In addition to the conventional CBN static part,² the language nPCF² has 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 \mathbb{b} , called the literals, can be “lifted” into the code types $\bigcirc\mathbb{b}$ with $\$_{\mathbb{b}}$, 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).

²We 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 $\tau ::= \mathbb{b} \mid \bigcirc\sigma \mid \tau_1 \rightarrow \tau_2$ (two-level types)
 $\sigma ::= \mathbb{b} \mid \sigma_1 \rightarrow \sigma_2$ (object-code types)
Contexts $\Gamma ::= \cdot \mid \Gamma, x : \tau$
Raw terms $E ::= \ell \mid x \mid \lambda x.E \mid E_1 E_2 \mid \mathbf{fix} E \mid \mathbf{if} E_1 E_2 E_3$
 $\mid E_1 \otimes E_2 \mid \$_{\mathbb{b}} E \mid \underline{d} \mid \lambda x.E \mid E_1 \underline{\otimes} E_2$

Typing Judgment $\boxed{\text{nPCF}^2 \vdash \Gamma \triangleright E : \tau}$

(Static)

$$\frac{\ell \in \mathbb{L}(\mathbb{b}) \quad x : \tau \in \Gamma}{\Gamma \triangleright \ell : \mathbb{b} \quad \Gamma \triangleright x : \tau}$$

$$\frac{\Gamma, x : \tau_1 \triangleright E : \tau_2 \quad \Gamma \triangleright E_1 : \tau_2 \rightarrow \tau \quad \Gamma \triangleright E_2 : \tau_2}{\Gamma \triangleright \lambda x.E : \tau_1 \rightarrow \tau_2 \quad \Gamma \triangleright E_1 E_2 : \tau}$$

$$\frac{\Gamma \triangleright E : \tau \rightarrow \tau \quad \Gamma \triangleright E_1 : \text{bool} \quad \Gamma \triangleright E_2 : \tau \quad \Gamma \triangleright E_3 : \tau}{\Gamma \triangleright \mathbf{fix} E : \tau \quad \Gamma \triangleright \mathbf{if} E_1 E_2 E_3 : \tau}$$

$$\frac{\Gamma \triangleright E_1 : \mathbb{b}_1 \quad \Gamma \triangleright E_2 : \mathbb{b}_2}{\Gamma \triangleright E_1 \otimes E_2 : \mathbb{b}} (\otimes : \mathbb{b}_1 \times \mathbb{b}_2 \rightarrow \mathbb{b})$$

(Dynamic)

$$\frac{\Gamma \triangleright E : \mathbb{b} \quad Sg(d) = \sigma \quad \Gamma, x : \bigcirc\sigma_1 \triangleright E : \bigcirc\sigma_2}{\Gamma \triangleright \$_{\mathbb{b}} E : \bigcirc\mathbb{b} \quad \Gamma \triangleright \underline{d} : \bigcirc\sigma \quad \Gamma \triangleright \lambda x.E : \bigcirc(\sigma_1 \rightarrow \sigma_2)}$$

$$\frac{\Gamma \triangleright E_1 : \bigcirc(\sigma_2 \rightarrow \sigma) \quad \Gamma \triangleright E_2 : \bigcirc\sigma_2}{\Gamma \triangleright E_1 \underline{\otimes} E_2 : \bigcirc\sigma}$$

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

Values $V ::= \ell \mid \lambda x.E \mid \mathcal{O}$
 $\mathcal{O} ::= \$_{\mathbb{b}} \ell \mid x \mid \lambda x.\mathcal{O} \mid \mathcal{O}_1 \underline{\otimes} \mathcal{O}_2 \mid \underline{d}$

(Static)

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

$$\frac{E(\mathbf{fix} E) \Downarrow V \quad E_1 \Downarrow \text{tt} \quad E_2 \Downarrow V \quad E_1 \Downarrow \text{ff} \quad E_3 \Downarrow V}{\mathbf{fix} E \Downarrow V \quad \mathbf{if} E_1 E_2 E_3 \Downarrow V \quad \mathbf{if} E_1 E_2 E_3 \Downarrow V}$$

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

(Dynamic)

$$\frac{E \Downarrow \ell}{\$_{\mathbb{b}} E \Downarrow \$_{\mathbb{b}} \ell} \quad \frac{x \Downarrow x \quad \underline{d} \Downarrow \underline{d}}{\lambda x.E \Downarrow \lambda x.\mathcal{O}} \quad \frac{E \Downarrow \mathcal{O} \quad E_1 \Downarrow \mathcal{O}_1 \quad E_2 \Downarrow \mathcal{O}_2}{E_1 \underline{\otimes} E_2 \Downarrow \mathcal{O}_1 \underline{\otimes} \mathcal{O}_2}$$

Figure 2: The two-level call-by-name language nPCF²

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

Typing Judgment $\boxed{\text{nPCF} \vdash \Delta \triangleright E : \sigma}$

The static part of nPCF² plus: $\frac{Sg(d) = \sigma}{\Delta \triangleright d : \sigma}$

Equational Rules $\boxed{\text{nPCF} \vdash \Delta \triangleright E_1 = E_2 : \sigma}$

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

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

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 evaluation can contain free dynamic variables [34]. 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 $\bigcirc\Delta \triangleright E : \tau$ and $E \Downarrow V$, then $\bigcirc\Delta \triangleright V : \tau$. ($\bigcirc\Delta$ is the element-wise application of the $\bigcirc(-)$ constructor to the context Δ .)*

As a consequence of this theorem, if $\bigcirc\Delta \triangleright E : \bigcirc\sigma$ 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 nPCF². 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 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 [30] of Plotkin's original transformation [42] maps a term E directly into a one-level term $\llbracket E \rrbracket_{\rho\kappa}$ (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 $\llbracket E \rrbracket_{df\kappa}$ (Figure 4c); evaluating $\llbracket E \rrbracket_{df\kappa}$ produces the resulting CPS term. The potential administrative redexes are annotated as static, and thus are reduced during the evaluation of $\llbracket E \rrbracket_{df\kappa}$. Intuitively, the one-pass transformation is derived by staging the program $\llbracket E \rrbracket_{\rho\kappa}$ [7].

By the definition of the translation, the two-level program $\llbracket E \rrbracket_{df\kappa}$ 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

³The call-by-name CPS transformation is studied in Appendix A of the full version of this paper [52].

⁴Less 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 $v\lambda$
Types $\sigma ::= \mathbf{b} \mid \sigma_1 \rightarrow \sigma_2$
Raw terms $E ::= x \mid \lambda x. E \mid E_1 E_2$
Typing judgment $\boxed{v\lambda \vdash \Delta \triangleright E : \sigma}$ (omitted)

b. Plotkin's original transformation:

$$\boxed{v\lambda \vdash \Delta \triangleright E : \sigma} \Rightarrow \boxed{\text{nPCF} \vdash \llbracket \Delta \rrbracket_{\rho\kappa} \triangleright \llbracket E \rrbracket_{\rho\kappa} : K \llbracket \sigma \rrbracket_{\rho\kappa}}$$

Here, $K\sigma = (\sigma \rightarrow \text{Ans}) \rightarrow \text{Ans}$ for an answer type Ans .
Types: $\llbracket \mathbf{b} \rrbracket_{\rho\kappa} = \mathbf{b}$, $\llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket_{\rho\kappa} = \llbracket \sigma_1 \rrbracket_{\rho\kappa} \rightarrow K \llbracket \sigma_2 \rrbracket_{\rho\kappa}$,
Terms: $\llbracket x \rrbracket_{\rho\kappa} = \lambda k. k x$, $\llbracket \lambda x. E \rrbracket_{\rho\kappa} = \lambda k. k \lambda x. \llbracket E \rrbracket_{\rho\kappa}$,
 $\llbracket E_1 E_2 \rrbracket_{\rho\kappa} = \lambda k. \llbracket E_1 \rrbracket_{\rho\kappa} \lambda r_1. \llbracket E_2 \rrbracket_{\rho\kappa} \lambda r_2. r_1 r_2 k$.

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

$$\boxed{v\lambda \vdash \Delta \triangleright E : \sigma} \Rightarrow \boxed{\text{nPCF}^2 \vdash \bigcirc \llbracket \Delta \rrbracket_{\rho\kappa} \triangleright \llbracket E \rrbracket_{df\kappa} : K^\bigcirc(\bigcirc \llbracket \sigma \rrbracket_{\rho\kappa})}$$

Here, $K^\bigcirc\tau = (\tau \rightarrow \bigcirc\text{Ans}) \rightarrow \bigcirc\text{Ans}$.

Terms:
 $\llbracket x \rrbracket_{df\kappa} = \lambda k. k x$,
 $\llbracket \lambda x. E \rrbracket_{df\kappa} = \lambda k. k \lambda x. \lambda k'. \llbracket E \rrbracket_{df\kappa} \lambda m. k' @m$,
 $\llbracket E_1 E_2 \rrbracket_{df\kappa} = \lambda k. \llbracket E_1 \rrbracket_{df\kappa} \lambda r_1. \llbracket E_2 \rrbracket_{df\kappa} \lambda r_2. r_1 @r_2 @ \underline{\lambda}. k a$.

The complete translation

$$\Rightarrow \boxed{\text{nPCF}^2 \vdash \bigcirc \llbracket \Delta \rrbracket_{\rho\kappa} \triangleright \llbracket E \rrbracket_{df\kappa} : \bigcirc(K \llbracket \sigma \rrbracket_{\rho\kappa})}$$

$$\llbracket E \rrbracket_{df\kappa} = \underline{\lambda} k. \llbracket E \rrbracket_{df\kappa} \lambda m. k @m$$

Figure 4: Call-by-value CPS transformation

the output of Plotkin's original transformation, $\llbracket E \rrbracket_{\rho\kappa}$. An intuitive argument is that erasing the annotations in $\llbracket E \rrbracket_{df\kappa}$ produces a term which is $\beta\eta$ -equivalent to $\llbracket E \rrbracket_{\rho\kappa}$.

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 nPCF² 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 nPCF²-phrase is the nPCF-phrase given as follows.*

Types: $|\bigcirc\sigma| = \sigma$, $|\mathbf{b}| = \mathbf{b}$, $|\tau_1 \rightarrow \tau_2| = |\tau_1| \rightarrow |\tau_2|$.
Terms: $|x| = x$, $|\$b E| = E$, $|\underline{d}| = d$, $|\underline{\lambda}x. E| = \lambda x. |E|$,
 $|E_1 @ E_2| = |E_1| |E_2|$.

Erasure of the static term constructs is homomorphic (e.g., $|\mathbf{fix} E| = \mathbf{fix} |E|$, $|\lambda x. E| = \lambda x. |E|$). If $\text{nPCF}^2 \vdash \Gamma \triangleright E : \tau$, then $\text{nPCF} \vdash |\Gamma| \triangleright |E| : |\tau|$. 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 nPCF² respects the nPCF-equality under erasure.

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

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

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 $v\Lambda \vdash \Delta \triangleright E : \sigma$ and $nPCF^2 \vdash \llbracket E \rrbracket_{dfc} \Downarrow \mathcal{O}$ then $nPCF \vdash \llbracket \Delta \rrbracket_{p\kappa} \triangleright |\mathcal{O}| = \llbracket E \rrbracket_{p\kappa} : K \llbracket \sigma \rrbracket_{p\kappa}$.*

Proof. A simple induction on E establishes that $nPCF \vdash \llbracket \Delta \rrbracket_{p\kappa} \triangleright |\llbracket E \rrbracket_{dfc}| = \llbracket E \rrbracket_{p\kappa} : K \llbracket \sigma \rrbracket_{p\kappa}$, which has the immediate corollary that $nPCF \vdash \llbracket \Delta \rrbracket_{p\kappa} \triangleright |\llbracket E \rrbracket_{dfc}| = \llbracket E \rrbracket_{p\kappa} : K \llbracket \sigma \rrbracket_{p\kappa}$. We then apply Theorem 2.3. \square

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 $nPCF^\Lambda$, 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):

$$\Lambda = \text{VAR of int} \mid \text{LIT}_b \text{ of } b \mid \text{CST of const} \\ \mid \text{LAM of int} \times \Lambda \mid \text{APP of } \Lambda \times \Lambda$$

The type const provides a representation $\{d\}$ 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 $\mathcal{D}(V)$:

$$\mathcal{D}(\text{VAR}(i)) = v_i, \quad \mathcal{D}(\text{LIT}_b(\ell)) = \ell, \quad \mathcal{D}(\text{CST}(\{d\})) = d \\ \mathcal{D}(\text{LAM}(i, e)) = \lambda v_i. \mathcal{D}(e), \quad \mathcal{D}(\text{APP}(e_1, e_2)) = \mathcal{D}(e_1) \mathcal{D}(e_2)$$

The language $nPCF^\Lambda$ has a standard, domain-theoretical CBN denotational semantics [31],⁵ which interprets the types as follows:

$$\llbracket \text{int} \rrbracket = \mathbf{Z}_\perp, \llbracket \text{bool} \rrbracket = \mathbf{B}_\perp, \llbracket \Lambda \rrbracket = \mathbf{E}_\perp, \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket = \llbracket \sigma_1 \rrbracket \rightarrow \llbracket \sigma_2 \rrbracket$$

⁵This is different from a lazy CBN semantics [51], which models Haskell-like languages where higher-order types are observable; there, function spaces are lifted, and the η -rule does not hold.

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 of the full version of this paper [52]), we remark that (1) the equational theory is sound with respect to the denotational semantics: If $nPCF^\Lambda \vdash \Delta \triangleright E_1 = E_2 : \sigma$, then $\llbracket E_1 \rrbracket = \llbracket E_2 \rrbracket$, and (2) the evaluation function for closed terms of base types induced from the denotational semantics has (by its computational adequacy with respect to an 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 [43]) efficient implementations that do not perform capture-avoiding substitutions.

Definition 2.5 (Embedding of $nPCF^2$ into $nPCF^\Lambda$: $\llbracket - \rrbracket_{nc}$).

$$\text{Types : } \llbracket \text{int} \rrbracket_{nc} = \text{int} \rightarrow \Lambda, \llbracket b \rrbracket_{nc} = b, \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket_{nc} = \llbracket \tau_1 \rrbracket_{nc} \rightarrow \llbracket \tau_2 \rrbracket_{nc} \\ \text{Terms : } \llbracket \$b E \rrbracket_{nc} = \$b^{nc} \llbracket E \rrbracket_{nc}, \llbracket \{d\} \rrbracket_{nc} = \lambda i. \text{CST}(\{d\}), \\ \llbracket \lambda x. E \rrbracket_{nc} = \lambda^{nc} \lambda x. \llbracket E \rrbracket_{nc}, \\ \llbracket E_1 \underline{\text{Q}} E_2 \rrbracket_{nc} = \underline{\text{Q}}^{nc} \llbracket E_1 \rrbracket_{nc} \llbracket E_2 \rrbracket_{nc}$$

where we use the following terms:

$$\$b^{nc} \equiv \lambda l. \lambda i. \text{LIT}_b(l), \\ \lambda^{nc} \equiv \lambda f. \lambda i. \text{LAM}(i, f(\lambda i'. \text{VAR}(i))(i + 1)), \\ \underline{\text{Q}}^{nc} \equiv \lambda m. \lambda n. \lambda i. \text{APP}(mi, ni).$$

Static constructs are translated homomorphically.

The three terms used in the embedding translation are $nPCF^\Lambda$ -terms themselves, and are kept as-is in the result of the translation. For instance, $\llbracket f \underline{\text{Q}} x \rrbracket_{nc}$ is the term $\underline{\text{Q}}^{nc} f x \equiv (\lambda m. \lambda n. \lambda i. \text{APP}(mi, ni)) f x$, not the simplified term $\lambda i. f i, x i$. 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 $nPCF^\Lambda$.

The translation uses a de Bruijn-level encoding for generating variable bindings. In addition, a dynamic λ -abstraction is translated using a static λ -abstraction⁶ and thus the two terms have the same binding behavior—a fact reflected in the following substitution lemma.

Lemma 2.6 (Substitution lemma for $\llbracket - \rrbracket_{nc}$). *If $nPCF^2 \vdash \Gamma, x : \tau' \triangleright E : \tau$ and $nPCF^2 \vdash \Gamma \triangleright E' : \tau'$, then $\llbracket E \{E'/x\} \rrbracket_{nc} \sim_\alpha \llbracket E \rrbracket_{nc} \llbracket \llbracket E' \rrbracket_{nc}/x \rrbracket$.*

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

Lemma 2.7 (Evaluation preserves translation). *If $nPCF^2 \vdash \text{int} \triangleright E : \tau$ and $nPCF^2 \vdash E \Downarrow V$, then $nPCF^\Lambda \vdash \llbracket \text{int} \rrbracket_{nc} \triangleright \llbracket E \rrbracket_{nc} = \llbracket V \rrbracket_{nc} = \llbracket \tau \rrbracket_{nc}$.*

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

⁶This is an instance of higher-order abstract syntax [41]. 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.

Lemma 2.8 (Translation of code-typed value). *If $\text{nPCF}^2 \vdash v_1 : \bigcirc\sigma_1, \dots, v_n : \bigcirc\sigma_n \triangleright \mathcal{O} : \bigcirc\sigma$, then there is a value $t : \Lambda$ such that $\text{nPCF}^\Lambda \vdash \triangleright (\llbracket \mathcal{O} \rrbracket_{\text{nc}}(n+1)) \{ \lambda i. \text{VAR}(1)/v_1, \dots, \lambda i. \text{VAR}(n)/v_n \} = t : \Lambda$, $\text{nPCF} \vdash v_1 : \sigma_1, \dots, v_n : \sigma_n \triangleright \mathcal{D}(t) : \sigma$, and $|\mathcal{O}| \sim_\alpha \mathcal{D}(t)$.*

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

Lemma 2.9 (Computational adequacy). *If $\text{nPCF}^2 \vdash \triangleright E : \bigcirc\sigma$, and there is a nPCF^Λ -value $t : \Lambda$ such that $\llbracket \llbracket E \rrbracket_{\text{nc}}(1) \rrbracket = \llbracket t \rrbracket$, 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 $\llbracket E \rrbracket_{\text{nc}}$. The definition of the logical relation at the type $\bigcirc\sigma$ implies the conclusion. \square

Theorem 2.10 (Correctness of embedding). *If $\text{nPCF}^2 \vdash \triangleright E : \bigcirc\sigma$, then the following statements are equivalent.*

- (a) *There is a value $\mathcal{O} : \bigcirc\sigma$ such that $\text{nPCF}^2 \vdash E \Downarrow \mathcal{O}$.*
 - (b) *There is a value $t : \Lambda$ such that $\llbracket \llbracket E \rrbracket_{\text{nc}}(1) \rrbracket = \llbracket t \rrbracket$.*
- When these statements hold, we further have that*
- (c) *$\text{nPCF} \vdash \triangleright \mathcal{D}(t) : \sigma$ 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 $\text{nPCF}^\Lambda \vdash \triangleright \llbracket E \rrbracket_{\text{nc}}(1) = t : \Lambda$, 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 $\text{nPCF}^{\text{tdpe}}$. Here, by a native normalization process, we mean a normalization algorithm that is implemented through a native embedding from $\text{nPCF}^{\text{tdpe}}$ into the implementation language.

The syntax of $\text{nPCF}^{\text{tdpe}}$ is displayed in Figure 5a. The language $\text{nPCF}^{\text{tdpe}}$ differs from nPCF^2 in that only base types

a. The language of CBN TDPE: $\text{nPCF}^{\text{tdpe}}$
 Type $\varphi ::= \mathbf{b} \mid \mathbf{b}^\mathfrak{d} \mid \varphi_1 \rightarrow \varphi_2$
 Raw terms $E ::= x \mid \ell \mid \lambda x. E \mid E_1 E_2 \mid \mathbf{fix} E_1 \mid \mathbf{if} E_1 E_2 E_3 \mid E_1 \otimes E_2 \mid \mathfrak{S}_b E \mid d^\mathfrak{d}$
 Typing judgment: $\boxed{\text{nPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi}$
 Typing rules: same as those of nPCF^2 , with the dynamic ones replaced by

$$\frac{Sg(d) = \sigma}{\Gamma \triangleright d^\mathfrak{d} : \sigma^\mathfrak{d}} \quad \frac{\Gamma \triangleright E : \mathbf{b}}{\Gamma \triangleright \mathfrak{S}_b E : \mathbf{b}^\mathfrak{d}},$$

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

b. Standard instantiation (TDPE-erasure)

$$\boxed{\text{nPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi} \implies \boxed{\text{nPCF} \vdash |\Gamma| \triangleright |E| : |\varphi|}$$

$$|\mathbf{b}^\mathfrak{d}| = \mathbf{b}; \quad |\mathfrak{S}_b E| = |E|, \quad |d^\mathfrak{d}| = d$$

c. Extraction functions \downarrow^σ and \uparrow_σ :

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

$$\begin{cases} \text{nPCF}^2 \vdash \triangleright \downarrow^\sigma : \sigma^\bigcirc \rightarrow \bigcirc\sigma \\ \downarrow^\mathbf{b} = \lambda x. x \\ \downarrow^{\sigma_1 \rightarrow \sigma_2} = \lambda f. \lambda x. \downarrow^{\sigma_2}(f(\uparrow_{\sigma_1} x)) \end{cases} \quad \begin{cases} \text{nPCF}^2 \vdash \triangleright \uparrow_\sigma : \bigcirc\sigma \rightarrow \sigma^\bigcirc \\ \uparrow_\mathbf{b} = \lambda x. x \\ \uparrow_{\sigma_1 \rightarrow \sigma_2} = \lambda e. \lambda x. \uparrow_{\sigma_2}(e \underline{\underline{e}}(\downarrow^{\sigma_1} x)) \end{cases}$$

d. Residualizing instantiation

$$\boxed{\text{nPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi} \implies \boxed{\text{nPCF}^2 \vdash \llbracket \Gamma \rrbracket_{ri} \triangleright \llbracket E \rrbracket_{ri} : \llbracket \varphi \rrbracket_{ri}}$$

$$\llbracket \mathbf{b}^\mathfrak{d} \rrbracket_{ri} = \bigcirc \mathbf{b}; \quad \llbracket \mathfrak{S}_b E \rrbracket_{ri} = \mathfrak{S}_b \llbracket E \rrbracket_{ri}, \quad \llbracket d^\mathfrak{d} : \sigma^\mathfrak{d} \rrbracket_{ri} = \uparrow_{\sigma^\mathfrak{d}} \underline{\underline{d}}$$

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

$$NF(E : \sigma^\mathfrak{d}) = \downarrow^\sigma \llbracket E \rrbracket_{ri} : \bigcirc\sigma$$

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

are binding-time annotated as static (\mathbf{b}) or dynamic ($\mathbf{b}^\mathfrak{d}$, instead of $\bigcirc \mathbf{b}$, 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 $\text{eq} : \text{int} \rightarrow \text{int} \rightarrow \text{bool}$, the corresponding dynamic constant is $\text{eq}^\mathfrak{d} : \text{int}^\mathfrak{d} \rightarrow \text{int}^\mathfrak{d} \rightarrow \text{bool}^\mathfrak{d}$; consequently, for what we write $\underline{\underline{\text{eq}}}(\mathfrak{S}_{\text{int}}(1+2)) : \bigcirc(\text{int} \rightarrow \text{bool})$ in nPCF^2 , we write $\text{eq}^\mathfrak{d}(\mathfrak{S}_{\text{int}}(1+2)) : \text{int}^\mathfrak{d} \rightarrow \text{bool}^\mathfrak{d}$ in $\text{nPCF}^{\text{tdpe}}$. Let us stress that there is no binding-time annotation for applications here.

The semantics is described through a *standard instanti-*

ation⁷ into the one-level language nPCF (Figure 5b), which amounts to erasing all the annotations; thus we overload the notation of erasure here.

Normalizing a closed nPCF^{tdpe}-term E of fully dynamic type σ^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 $\text{nPCF} \vdash \triangleright |E| = E' : \sigma$. For example, normalizing the term $\text{eq}^d (\$_{\text{int}}(1+2))$ should produce the object term $\lambda x.\text{eq}3x$. 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 nPCF², is shown in Figure 5c-e. It finds the normal form of a nPCF^{tdpe}-term $E : \sigma^d$ by applying a type-indexed extraction function \downarrow^σ (“reification”) on a particular instantiation, called the *residualizing instantiation* $\{\!\!|E\!\!\}_i$, of term E in the language nPCF². Being an instantiation, which maps static constructs to themselves, $\{\!\!|E\!\!\}_i$ makes the TDPE algorithm naturally implementable in nPCF² through the embedding $\{\!\!|-\!\!\}_{\text{nc}}$ of Section 2.4. Indeed, the composition of $\{\!\!|-\!\!\}_i$ and the embedding $\{\!\!|-\!\!\}_{\text{nc}}$ 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 , $\text{nPCF} \vdash \triangleright |\downarrow^\sigma| = \lambda x.x : \sigma \rightarrow \sigma$ and $\text{nPCF} \vdash \triangleright |\uparrow_\sigma| = \lambda x.x : \sigma \rightarrow \sigma$.*

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 $\text{nPCF}^{\text{tdpe}} \vdash \triangleright E : \sigma^d$ and $\text{nPCF}^2 \vdash NF(E) \downarrow \mathcal{O}$, then $\text{nPCF} \vdash \triangleright |\mathcal{O}| = |E| : \sigma$.*

Proof. A simple induction on E establishes that $\text{nPCF} \vdash \triangleright |\{\!\!|E\!\!\}_i| = |E| : \sigma$, which has the immediate corollary that $\text{nPCF} \vdash \triangleright |NF(E)| = |E| : \sigma$. We then apply Theorem 2.3. \square

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

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

the output is fully η -expanded by typing, however, appears to require dependent types for the term representation.⁸

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 nPCF²-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 nPCF², 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 $\bigcirc^{\text{var}}(-)$ 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.

$$\frac{\Delta \triangleright^{\text{at}} E : \mathbf{b} \quad \Delta, x : \sigma_1 \triangleright^{\text{nf}} E : \sigma_2 \quad \ell \in \mathbb{L}(\mathbf{b})}{\Delta \triangleright^{\text{nf}} E : \mathbf{b} \quad \Delta \triangleright^{\text{nf}} \lambda x.E : \sigma_1 \rightarrow \sigma_2 \quad \Delta \triangleright^{\text{at}} \ell : \mathbf{b}}$$

$$\frac{Sg(d) = \sigma \quad x : \sigma \in \Delta \quad \Delta \triangleright^{\text{at}} E_1 : \sigma_2 \rightarrow \sigma \quad \Delta \triangleright^{\text{nf}} E_2 : \sigma_2}{\Delta \triangleright^{\text{at}} d : \sigma \quad \Delta \triangleright^{\text{at}} x : \sigma \quad \Delta \triangleright^{\text{at}} E_1 E_2 : \sigma}$$

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

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

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

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

(Dynamic)

$$\frac{\Gamma \blacktriangleright E : \bigcirc^{\text{at}}(\mathbf{b}) \quad \Gamma, x : \bigcirc^{\text{var}}(\sigma_1) \blacktriangleright E : \bigcirc^{\text{nf}}(\sigma_2)}{\Gamma \blacktriangleright E : \bigcirc^{\text{nf}}(\mathbf{b}) \quad \Gamma \blacktriangleright \lambda x.E : \bigcirc^{\text{nf}}(\sigma_1 \rightarrow \sigma_2)}$$

$$\frac{\Gamma \blacktriangleright E : \mathbf{b} \quad Sg(d) = \sigma}{\Gamma \blacktriangleright \$_{\mathbf{b}}E : \bigcirc^{\text{at}}(\mathbf{b}) \quad \Gamma \blacktriangleright \underline{d} : \bigcirc^{\text{at}}(\sigma)}$$

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

Figure 7: nPCF²-terms that generate code in long $\beta\eta$ -normal form

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

⁸On 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].

Theorem 2.14 (Normal-form code types). *If V is an nPCF^2 -value (Figure 2), then*

- (1) *if $\text{nPCF}^2 \vdash \bigcirc^{\text{var}}(\Delta) \blacktriangleright V : \bigcirc^{\text{at}}(\sigma)$, then $V \equiv \mathcal{O}$ for some \mathcal{O} and $\Delta \triangleright^{\text{at}} |\mathcal{O}| : \sigma$;*
- (2) *if $\text{nPCF}^2 \vdash \bigcirc^{\text{var}}(\Delta) \blacktriangleright V : \bigcirc^{\text{nf}}(\sigma)$, then $V \equiv \mathcal{O}$ for some \mathcal{O} and $\Delta \triangleright^{\text{nf}} |\mathcal{O}| : \sigma$.*

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 $\sigma^{\bigcirc^{\text{nf}}}$ for $\sigma\{\bigcirc^{\text{nf}}(b)/b : b \in \mathbb{B}\}$).*

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

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

Theorem 2.16. *If $\text{nPCF}^{\text{tdpe}} \vdash \triangleright E : \sigma^\circ$, then $\text{nPCF}^2 \vdash \blacktriangleright \text{NF}(E) : \bigcirc^{\text{nf}}(\sigma)$.*

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

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, 44]. 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 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 L^2 (e.g., 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 L^2 , we identify and prove properties of 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: $L \vdash g = |E|$. 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

Let us turn to design a two-level language vPCF^2 with Moggi’s computational λ -calculus λ_c [32] as its object language, taking into consideration the desired properties that we identified in Section 3.

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 $E_2 \triangleq \text{print}@\$(\text{int}(2+2))$, where $\text{print} : \bigcirc(\text{int} \rightarrow \text{bool})$ is a dynamic constant. Then

the code generated by the program $(\lambda x.\mathbf{let} y \leftarrow x \mathbf{in} x) E_2$ after erasure would be $\mathbf{let} y \leftarrow (\mathbf{print} 4) \mathbf{in} (\mathbf{print} 4)$, which incorrectly duplicates the computation $\mathbf{print} 4$.

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 $\mathbb{V}\sigma$ 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 $\mathbb{E}\sigma$ to indicate possible computational effects, can be coerced into type $\mathbb{V}\sigma$ with a “trivialization” operator $\#$, which performs let-insertion.

The syntax, the semantics, and some key properties of $vPCF^2$, together with its application in formalizing and implementing call-by-value TDPE, appear in the appendix.

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.

5.1 Two-level formalisms for compiler construction

While Jones et al. [21, 22, 23] 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 the associated analysis algorithms [37]. 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 [48]. 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 [38], provides a setting for comparing and relating such languages, and, on the prescriptive side [39], 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 [27], viewed as an interpreter, into a compiler [35]. Mosses developed action semantics [36] 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 [29].

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 [33]. 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 [49, 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 [40] 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 [19]. 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 [25, 26], 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 [24].

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,⁹ 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 [50].

⁹It would be, however, interesting to see whether real-life applications like the automake suite in Unix can be described as three-level programs.

Davies and Pfenning investigated multi-level languages through the Curry-Howard correspondence with modal logics: λ^\square , which corresponds to intuitionistic modal logic S4, has the run construct, but it can only manipulate closed code fragment [11]; λ° , 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 [34] combines λ^\square and λ° , 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 λ^\square in terms of an abstract machine [50]. Taha axiomatized a fragment of Idealized MetaML, which can be used for equational reasoning [47].

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).

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 transformation), 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 $vPCF^{\wedge, st}$, 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 A.2) 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, 34, 37].
- 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.

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 Calcagno, 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 [45], 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 [45], 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 [28], pages 22–32. Extended version to appear in the journal *Higher-Order and Symbolic Computation*, 14(2/3), 2001.
- [18] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.
- [19] John McCarthy *et al.* *LISP 1.5 Programmer’s Manual*. The MIT Press, Cambridge, Massachusetts, 1962.
- [20] Neil D. Jones, editor. *Special issue on Partial Evaluation*, *Journal of Functional Programming*, Vol. 3, Part 3. Cambridge University Press, July 1993.
- [21] 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>.
- [22] 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.
- [23] 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.
- [24] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised⁵ 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/>.
- [25] 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.
- [26] 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.
- [27] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.
- [28] 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.
- [29] Peter Lee. *Realistic Compiler Generation*. The MIT Press, 1989.
- [30] 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.
- [31] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.
- [32] 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.
- [33] 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.
- [34] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaïssa, 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.
- [35] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

- [36] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.
- [37] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.
- [38] 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.
- [39] Flemming Nielson and Hanne Riis Nielson. Prescriptive frameworks for multi-level lambda-calculi. In Charles Conzel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 193–202, Amsterdam, The Netherlands, June 1997.
- [40] Jens Palsberg. Correctness of binding-time analysis. In Jones [20], pages 347–363.
- [41] 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.
- [42] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [43] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.
- [44] 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.
- [45] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.
- [46] Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approach. In Lawall [28], 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.
- [47] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trivial (extended abstract). In Lawall [28], pages 34–43.
- [48] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.
- [49] Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones [20], pages 365–387.
- [50] 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.
- [51] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.
- [52] Zhe Yang. Reasoning about code generation in two-level languages (extended version, 74 pages). Technical Report BRICS RS-00-46, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 2000. Available at <http://cs.nyu.edu/zheyang/papers/Reason2Level-long.ps.gz>.

Types	$\tau ::= \theta \mid \textcircled{\sigma}$ $\theta ::= \mathbf{b} \mid \textcircled{\vee}\sigma \mid \theta \rightarrow \tau$ (substitution-safe types) $\sigma ::= \mathbf{b} \mid \sigma_1 \rightarrow \sigma_2$ (object-code types)
Contexts	$\Gamma ::= \cdot \mid \Gamma, x : \theta$
Raw terms	$E ::= \ell \mid x \mid \lambda x. E \mid E_1 E_2 \mid \mathbf{fix} E \mid \mathbf{if} E_1 E_2 E_3$ $\mid E_1 \otimes E_2 \mid \textcircled{\$}_b E \mid \underline{\lambda} x. E \mid E_1 \underline{\textcircled{E}}_2 \mid \underline{d}$ $\mid \mathbf{let} x \leftarrow E_1 \mathbf{in} E_2 \mid \#E$
Typing Judgment	$\text{vPCF}^2 \vdash \Gamma \triangleright E : \tau$
(Static)	$\frac{\ell \in \mathbb{L}(\mathbf{b}) \quad x : \theta \in \Gamma \quad \Gamma, x : \theta_1 \triangleright E : \tau_2}{\Gamma \triangleright \ell : \mathbf{b} \quad \Gamma \triangleright x : \theta \quad \Gamma \triangleright \lambda x. E : \theta_1 \rightarrow \tau_2}$ $\frac{\Gamma \triangleright E_1 : \theta_2 \rightarrow \tau \quad \Gamma \triangleright E_2 : \theta_2 \quad \Gamma \triangleright E : (\theta_1 \rightarrow \tau_2) \rightarrow (\theta_1 \rightarrow \tau_2)}{\Gamma \triangleright E_1 E_2 : \tau \quad \Gamma \triangleright \mathbf{fix} E : \theta_1 \rightarrow \tau_2}$ $\frac{\Gamma \triangleright E_1 : \mathbf{bool} \quad \Gamma \triangleright E_2 : \tau \quad \Gamma \triangleright E_3 : \tau}{\Gamma \triangleright \mathbf{if} E_1 E_2 E_3 : \tau}$ $\frac{\Gamma \triangleright E_1 : \mathbf{b}_1 \quad \Gamma \triangleright E_2 : \mathbf{b}_2}{\Gamma \triangleright E_1 \otimes E_2 : \mathbf{b}} (\otimes : \mathbf{b}_1 \times \mathbf{b}_2 \rightarrow \mathbf{b})$
(Dynamic)	$\frac{\Gamma \triangleright E : \mathbf{b} \quad \text{Sg}(d) = \sigma \quad \Gamma, x : \textcircled{\vee}\sigma_1 \triangleright E : \textcircled{\sigma}_2}{\Gamma \triangleright \textcircled{\$}_b E : \textcircled{\vee}\mathbf{b} \quad \Gamma \triangleright \underline{d} : \textcircled{\vee}\sigma \quad \Gamma \triangleright \underline{\lambda} x. E : \textcircled{\vee}(\sigma_1 \rightarrow \sigma_2)}$ $\frac{\Gamma \triangleright E_1 : \textcircled{\sigma}(\sigma_2 \rightarrow \sigma) \quad \Gamma \triangleright E_2 : \textcircled{\sigma}_2 \quad \Gamma \triangleright E : \textcircled{\vee}\sigma}{\Gamma \triangleright E_1 \underline{\textcircled{E}}_2 : \textcircled{\sigma} \quad \Gamma \triangleright E : \textcircled{\sigma}}$ $\frac{\Gamma, x : \textcircled{\vee}\sigma_1 \triangleright E_2 : \textcircled{\sigma}_2 \quad \Gamma \triangleright E_1 : \textcircled{\sigma}_1 \quad \Gamma \triangleright E : \textcircled{\sigma}}{\Gamma \triangleright \mathbf{let} x \leftarrow E_1 \mathbf{in} E_2 : \textcircled{\sigma} \quad \Gamma \triangleright \#E : \textcircled{\vee}\sigma}$

Figure 8: The type system of vPCF²

A The language vPCF²

A.1 Syntax, semantics, and properties

The syntax and evaluation semantics of vPCF² 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 $\text{if}_\sigma : \mathbf{bool} \rightarrow (\text{unit} \rightarrow \sigma) \rightarrow (\text{unit} \rightarrow \sigma)$ (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 $\textcircled{\sigma}$. 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 let and other constructs.

In the evaluation semantics of vPCF², the accumulated bindings B are explicit; furthermore, the dynamic environment Δ is necessary, because the generation 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 begin-

Values $V ::= \ell \mid \lambda x. E \mid \mathcal{O}$
 $\mathcal{O} ::= \$_b \ell \mid x \mid \lambda x. \mathcal{O} \mid \mathcal{O}_1 @ \mathcal{O}_2 \mid \underline{d} \mid \mathbf{let} \ x \leftarrow \mathcal{O}_1 \ \mathbf{in} \ \mathcal{O}_2$

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

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

We use the following abbreviations.

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

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

(Static)

$$\frac{\ell \Downarrow \ell \quad \lambda x. E \Downarrow \lambda x. E \quad \frac{E_1 \Downarrow \lambda x. E' \quad E_2 \Downarrow V' \quad E' \{V'/x\} \Downarrow V}{E_1 E_2 \Downarrow V}}{E \Downarrow \lambda x. E' \quad E' \{\mathbf{fix}(\lambda x. E')/x\} \Downarrow V \quad \frac{E_1 \Downarrow \mathbf{tt} \quad E_2 \Downarrow V}{\mathbf{if} \ E_1 \ E_2 \ E_3 \ \Downarrow V}}}{\frac{E_1 \Downarrow \mathbf{ff} \quad E_3 \Downarrow V \quad \frac{E_1 \Downarrow V_1 \quad E_2 \Downarrow V_2}{E_1 \otimes E_2 \Downarrow V} (V_1 \otimes V_2 = V)}{\mathbf{if} \ E_1 \ E_2 \ E_3 \ \Downarrow V}}$$

(Dynamic)

$$\frac{E \Downarrow \ell \quad \frac{\frac{\frac{E_1 \Downarrow \mathcal{O}_1 \quad E_2 \Downarrow \mathcal{O}_2}{E_1 @ E_2 \Downarrow \mathcal{O}_1 @ \mathcal{O}_2} \quad x \Downarrow x \quad \underline{d} \Downarrow \underline{d}}{\$_b E \Downarrow \$_b \ell} \quad \frac{\Delta \triangleright [B] \lambda x. E \Downarrow [B'] \mathcal{O} \quad y \notin \text{dom } B \cup \text{dom } \Delta}{\Delta \triangleright [B] \lambda x. E \Downarrow [B'] \lambda y. \mathbf{let}^* \ B' \ \mathbf{in} \ \mathcal{O}}}{\Delta \triangleright [B] E_1 \Downarrow [B'] \mathcal{O}_1 \quad \Delta \triangleright [B] E_2 \Downarrow [B'] \mathcal{O}_2 \quad y \notin \text{dom } B' \cup \text{dom } \Delta}}{\Delta \triangleright [B] \mathbf{let} \ y \leftarrow E_1 \ \mathbf{in} \ E_2 \Downarrow [B'] \mathbf{let} \ x \leftarrow \mathcal{O}_1 \ \mathbf{in} \ (\mathbf{let}^* \ B'' \ \mathbf{in} \ \mathcal{O}_2)}$$

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

Figure 9: The evaluation semantics of vPCF²

ning, 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. (Again, the detailed development can be found in the full version of this paper [52].)

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 [18].

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

Theorem A.2 (Type preservation). If $\mathbb{V}\Delta \triangleright [B]E : \tau$ and $\Delta \triangleright [B]E \Downarrow [B']V$, then $\mathbb{V}\Delta \triangleright [B']V : \tau$.

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

Definition A.3 (Observation of complete program). For a complete program $\triangleright E : \mathbb{C}\sigma$, we write $E \searrow \mathbf{let}^* \ B \ \mathbf{in} \ \mathcal{O}$ if $\triangleright [\cdot]E \Downarrow [B]\mathcal{O}$.

Corollary A.4 (Type preservation for complete programs). If $\triangleright E : \mathbb{C}\sigma$ and $E \searrow \mathcal{O}$, then $\triangleright \mathcal{O} : \mathbb{C}\sigma$.

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 A.5 (Annotation erasure). If $\text{vPCF}^2 \vdash \mathbb{V}\Delta \triangleright [B]E : \tau$ and $\text{vPCF}^2 \vdash \Delta \triangleright [B]E \Downarrow [B']V$, then $\text{vPCF} \vdash \Delta \triangleright \mathbf{let}^* \ |B| \ \mathbf{in} \ |E| = \mathbf{let}^* \ |B| \ \mathbf{in} \ |V| : |\tau|$.

Theorem A.6 (Annotation erasure for complete programs). If $\text{vPCF}^2 \vdash \triangleright E : \mathbb{C}\sigma$ and $\text{vPCF}^2 \vdash E \searrow \mathcal{O}$, then $\text{vPCF} \vdash \triangleright |E| = |\mathcal{O}| : \sigma$.

Native embedding: Without going into detail, we remark that vPCF² has a simple native embedding $\{\cdot\}_{\text{ve}}$ into vPCF^{Λ, 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 vPCF^{Λ, st} is a subset of ML; the language vPCF² can thus be embedded into ML, with dynamic constructs defined as functions.¹⁰ The correctness proof for the embedding is by directly relating the derivation of the evaluation from a term E , in vPCF², and the derivation of the evaluation from its translation $\{E\}_{\text{ve}}$, in vPCF^{Λ, st}. The details of the native embedding and the accompanying correctness proof, again, are available in Appendix C of the full version of this paper [52].

A.2 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 checks in vPCF². 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 $\{\cdot\}_{\text{ve}}$, we have an efficient implementation of this formulation—which is essentially the call-by-value TDPE algorithm that uses state-based let-insertion [46]; see also Filinski’s formal treatment [13].

¹⁰The ML source code, with the following example of CBV TDPE, is available at the URL www.brics.dk/~zheyang/programs/vPCF2.

- a. The language of CBV TDPE: $\text{vPCF}^{\text{tdpe}}$
The syntax is the same as that of CBN TDPE, with the addition of a let-construct.

$$\frac{\Delta, x : \sigma_1 \triangleright E_2 : \sigma_2 \quad \Delta \triangleright E_1 : \sigma_1}{\Delta \triangleright \text{let } x \leftarrow E_1 \text{ in } E_2 : \sigma_2}$$

- b. Standard instantiation (TDPE-erasure)

$$\boxed{\text{vPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi} \implies \boxed{\text{vPCF} \vdash |\Gamma| \triangleright |E| : |\varphi|}$$

$$|b^\flat| = b; \quad |\$bE| = |E|, |d^\flat| = d$$

- c. Extraction functions \downarrow^σ and \uparrow_σ :

We write σ^\flat for the type $\sigma(\mathbb{V}b/b : b \in \mathbb{B})$.

$$\begin{cases} \text{vPCF}^2 \vdash \triangleright \downarrow^\sigma : \sigma^\flat \rightarrow \mathbb{V}\sigma \\ \downarrow_b^b = \lambda x.x \\ \downarrow^{\sigma_1 \rightarrow \sigma_2} = \lambda f. \lambda x. \downarrow^{\sigma_2}(f(\uparrow_{\sigma_1} x)) \end{cases}$$

$$\begin{cases} \text{vPCF}^2 \vdash \triangleright \uparrow_\sigma : \mathbb{V}\sigma \rightarrow \sigma^\flat \\ \uparrow_b^b = \lambda x.x \\ \uparrow_{\sigma_1 \rightarrow \sigma_2} = \lambda e. \lambda x. \uparrow_{\sigma_2} \#(e \underline{\text{@}}(\downarrow^{\sigma_1} x)) \end{cases}$$

- d. Residualizing instantiation

$$\boxed{\text{vPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi} \implies \boxed{\text{vPCF}^2 \vdash \{\Gamma\}_{ri} \triangleright \{E\}_{ri} : \{\varphi\}_{ri}}$$

$$\{\!|b^\flat|\!\}_{ri} = \mathbb{V}b; \quad \{\!|\$bE|\!\}_{ri} = \$b\{E\}_{ri}, \{\!|d^\flat|\!\}_{ri} = \uparrow_\sigma d^\flat$$

- e. The static normalization function

$$NF(\triangleright E : \sigma^\flat) = \downarrow^\sigma \{\!|E|\!\}_{ri} : \textcircled{\sigma}$$

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 $\textcircled{\sigma}$.

Definition A.7 (Refined binder-term-in-context).

For a binder $B \equiv (x_1 : \sigma_1 = \mathcal{O}_1, \dots, x_n : \sigma_n = \mathcal{O}_n)$, we write $\Gamma \triangleright [B]E : \tau$ if $\Gamma, x_1 : \mathbb{V}^{var}(\sigma_1), \dots, x_{i-1} : \mathbb{V}^{var}(\sigma_{i-1}) \triangleright \mathcal{O}_i : \textcircled{\sigma_i}$ for all $1 \leq i \leq n$, and $\Gamma, x_1 : \mathbb{V}^{var}(\sigma_1), \dots, x_n : \mathbb{V}^{var}(\sigma_n) \triangleright E : \tau$.

Theorem A.8 (Refined type preservation). If $\text{vPCF}^2 \vdash$

$$\begin{aligned} \text{Types } \tau &::= \theta \mid \textcircled{\sigma}^{bd} \mid \textcircled{\sigma}^{nc} \\ \theta &::= b \mid \mathbb{V}^{var}(\sigma) \mid \mathbb{V}^{nv}(\sigma) \mid \mathbb{V}^{av}(\sigma) \mid \theta \rightarrow \tau \\ \sigma &::= b \mid \sigma_1 \rightarrow \sigma_2 \end{aligned}$$

Typing Judgment $\boxed{\text{vPCF}^2 \vdash \Gamma \triangleright E : \tau}$

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

(Dynamic)

$$\frac{\Gamma \triangleright E : \mathbb{V}^{var}(\sigma) \quad Sg(d) = \sigma \quad \Gamma \triangleright E : b}{\Gamma \triangleright E : \mathbb{V}^{av}(\sigma) \quad \Gamma \triangleright d : \mathbb{V}^{av}(\sigma) \quad \Gamma \triangleright \$bE : \mathbb{V}^{av}(b)}$$

$$\frac{\Gamma \triangleright E : \mathbb{V}^{av}(b) \quad \Gamma, x : \mathbb{V}^{var}(\sigma_1) \triangleright E : \textcircled{\sigma_2}^{nc}}{\Gamma \triangleright E : \mathbb{V}^{nv}(b) \quad \Gamma \triangleright \lambda x.E : \mathbb{V}^{nv}(\sigma_1 \rightarrow \sigma_2)}$$

$$\frac{\Gamma \triangleright E_1 : \mathbb{V}^{av}(\sigma_2 \rightarrow \sigma) \quad \Gamma \triangleright E_2 : \mathbb{V}^{nv}(\sigma_2) \quad \Gamma \triangleright E : \mathbb{V}^{nv}(\sigma)}{\Gamma \triangleright E_1 \underline{\text{@}} E_2 : \textcircled{\sigma}^{bd} \quad \Gamma \triangleright E : \textcircled{\sigma}^{nc}}$$

$$\frac{\Gamma, x : \mathbb{V}^{var}(\sigma_1) \triangleright E_2 : \textcircled{\sigma_2}^{nc} \quad \Gamma \triangleright E_1 : \textcircled{\sigma_1}^{bd}}{\Gamma \triangleright \text{let } x \leftarrow E_1 \text{ in } E_2 : \textcircled{\sigma}^{nc}}$$

$$\frac{\Gamma \triangleright E : \textcircled{\sigma}^{bd}}{\Gamma \triangleright \#E : \mathbb{V}^{var}(\sigma)}$$

Figure 11: vPCF^2 -terms that generate code in λ_c -normal form

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

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

$$\frac{\Delta \triangleright^{nv} E : \sigma \quad \Delta, x : \sigma_1 \triangleright^{nc} E_2 : \sigma_2 \quad \Delta \triangleright^{bd} E_1 : \sigma_1}{\Delta \triangleright^{nc} E : \sigma \quad \Delta \triangleright^{nc} \text{let } x \leftarrow E_1 \text{ in } E_2 : \sigma_2}$$

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

$\mathbb{V}^{var}(\Delta) \triangleright [B]E : \tau$ and $\text{vPCF}^2 \vdash \Delta \triangleright [B]E \downarrow [B']V$, then $\text{vPCF}^2 \vdash \mathbb{V}^{var}(\Delta) \triangleright [B']V : \tau$.

Corollary A.9 (Refined type preservation for complete programs). If $\triangleright E : \textcircled{\sigma}^{nc}$ and $E \searrow \mathcal{O}$, then $\triangleright \mathcal{O} : \textcircled{\sigma}^{nc}$.

Theorem A.10 (Normal-form code types). If V is an vPCF^2 -value (Figure 8), and $\text{vPCF}^2 \vdash \mathbb{V}^{var}(\Delta) \triangleright V : \mathbb{V}^X(\sigma)$ where X is av , nv , bd , or nc , then $V \equiv \mathcal{O}$ for some \mathcal{O} and $\Delta \triangleright^X |\mathcal{O}| : \sigma$.

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 A.11. (1) The extraction functions (Figure 10c) have the following normal-form types (writing $\sigma^{\text{O}nv}$ for $\sigma(\mathbb{V}^{nv}(b)/b : b \in \mathbb{B})$.)

$$\triangleright \downarrow^\sigma : \sigma^{\text{O}nv} \rightarrow \mathbb{V}^{nv}(\sigma), \triangleright \uparrow_\sigma : \mathbb{V}^{av}(\sigma) \rightarrow \sigma^{\text{O}nv}.$$

(2) If $\text{vPCF}^{\text{tdpe}} \vdash \Gamma \triangleright E : \varphi$, then $\text{vPCF}^2 \vdash \{\!|\varphi|\!\}_{ri}^{nv} \triangleright \{\!|E|\!\}_{ri}$: $\{\!|\varphi|\!\}_{ri}^{nv}$, where $\{\!|\varphi|\!\}_{ri}^{nv} = \varphi(\mathbb{V}^{nv}(b)/b^\flat : b \in \mathbb{B})$.

Theorem A.12. If $\text{vPCF}^{\text{tdpe}} \vdash \triangleright E : \sigma^\flat$, then $\text{vPCF}^2 \vdash NF(E) : \mathbb{V}^{nv}(\sigma)$.