Language Support

for Program Generation

Reasoning, Implementation, and Applications

by

Zhe Yang

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

Graduate School of Arts and Science

New York University

September 2001

Olivier Danvy

Benjamin Goldberg

In memory of Bob Paige

Acknowledgment

My graduate study and life, of which this dissertation is one of the outcomes, benefitted greatly from the time, energy, and enthusiasm of many people.

I am honored to have studied at both the Courant Institute and at the BRICS International PhD school, under the guidance of some most wonderful teachers. From both my former advisor, the late professor Bob Paige, and my PhD advisor Olivier Danvy, I learned to appreciate the beauty of mathematics—and of music as well—and to never lose sight of the practical impact. They both demonstrated great scholarship with devotion to science and also great care for their students. Their support has been boundless and timeless, available even during their most difficult times.

Andrzej Filinski and Fritz Henglein deserve a great deal of credit: their insights and rigor sharpened my understanding of programming languages and semantics. Benjamin Goldberg and my undergraduate thesis advisor Kai Lin introduced me to the art of programming languages, and helped me in many ways at different stages of my study. Neil D. Jones and Suresh Jagannathan hosted me for two valuable research trips, at DIKU and NEC Research. I would also like to thank Richard Cole, Fritz Henglein, Amir Pnueli, and Alan Siegel for serving on my thesis committee.

I am grateful to Deepak Goyal and Bernd Grobauer, both of who collaborated

closely with me on my research, and shared with me many enjoyable moments of life. I eagerly expect those memorable tennis matches and bike trips to repeat, just as I hope to work with them again. Xianghui Duan and Daniel Damian were two wonderful officemates. With Gabriel Juhás I endured a winter that seemed endless, yet passed so quickly. With David Tanzer I endured the initial bafflement of studying category theory; he wiped it away with his guitar tunes.

Both Courant Institute and BRICS have been enjoyable places to work. I would like to thank Rosemary Amico, Janne Christensen, Anina Karmen, and Karen Møller for their timely help. Richard Cole, Mogens Nielson, Glynn Winskel, Alan Siegel provided both outstanding lectures and much needed help. My student life was made an enjoyable and enriching experience by many friends, including Gedas Advomavicius, Emanuela Boem, Fangzhe Chang, Cheryl Chen, Hseu-Ming Chen, Stefan Dziembowski, Trung Dinh Dao, Alina Grabiec, Marcin Jurdzinski, Allen Leung, Chen Li, Ninghui Li, Madhu Nayakkankuppam, Ed Osinski, Magda Olbryt, Alexa Olesen, Toto Paxia, Archi Rudra, Simona Penna, Patricia Gil Rios, Alicja Marczak, Christian Ungureanu, and Daniele Varacca. The spirit of Courant Basketball Association, of the language-beer-jogging group at Courant, and of the badminton and soccer teams at both places has fueled my imagination in computer science. Three unforgettable stays hosted by three warm European families—of Jo Keller in Paris, of Bernd Grobauer in Deggendorf, and of Raffaella Grasso in Genova—were calm oases in an ever-tense student's life.

I want to thank my parents and my sister Delin Yang for cultivating in me an interest in computer science many years ago, for giving me the opportunity to explore this subject many years later, and for being patient during my graduate study, while providing the necessary pressure. Finally, I would like to thank Raffaella Grasso, for her love.

Abstract

This dissertation develops programming languages and associated techniques for sound and efficient implementations of algorithms for program generation.

First, we develop a framework for practical two-level languages. In this framework, we demonstrate that two-level languages are not only a good tool for describing program-generation algorithms, but a good tool for reasoning about them and implementing them as well. We pinpoint several general properties of two-level languages that capture common proof obligations of program-generation algorithms:

- To prove that the generated program that it behaves as desired, we use an *erasure* property to reduce the two-level proof obligation to a simpler one-level obligation.

- To prove that the generated program satisfies certain syntactic constraints, we use a *type-preservation* property for a refined type system that enforces these constraints.

In addition, to justify concrete implementations, 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, and demonstrate them through 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.

Next, to facilitate implementations, we develop several general approaches to programming with *type-indexed families* of values within the popular Hindley-Milner type system. Type-indexed families provide a form of type dependency, which is employed by many algorithms that generate typed programs, but is absent from mainstream languages. Our approaches are based on type encodings, so that they are type safe. We demonstrate and compare them through a host of examples, including type-directed partial evaluation and printf-style formatting.

Finally, upon the two-level framework and type-encoding techniques, we recast a joint work with Bernd Grobauer, where we formally derived a suitable self application for type-directed partial evaluation, and achieved automatic compiler generation.

Contents

List of Figures

List of Appendices

Chapter 1

Introduction

This dissertation aims to show that *practical language support for program generation can be developed, and that it should be developed both to address various demands from the application domain and to provide sound and tractable implementations.*

More specifically, while adopting two-level languages as the main vehicle for expressing code-generation algorithms, we identify the important properties that help programmers to debug and to reason about the code-generation algorithms, and then accordingly, to design two-level languages with these properties. In the meantime, we also make sure these languages do have effective implementations. To this end, we show that these languages can be effectively embedded into existing, mainstream programming languages, which are already endowed with efficient implementations.

1.1 Language-based approach in general

Software systems are becoming more complex, more diverse, and more distributed. Programming-language methods have found an increasing number of applications in software systems. Indeed, while modular organization is suitable for most applications, it is not uncommon that introducing some language abstraction helps greatly, because of, e.g., a set of notations that more succinctly and accurately models certain aspects of the computation, or a type system that prevents critical systems from abuse. Example applications of programming-language method include the FoxNet [34] and Anno Domini [28].

Typically, to address a class of applications with the language-based approach, we need to balance the need of expressiveness, of correctness proofs, and of implementation. We look in the application domain for the suitable notations that comprise the programming language: they should be expressive enough to cover a large class of applications, while restricted enough to bear semantic properties that help programmers to debug and to reason about these programs. In other words, in view of the understanding from the area of formal methods that one programs against a specification, the design of a programming language should take common proof obligations of its programs into account.

Furthermore, for a language or a language abstraction to be useful in programming practice, it is indispensable to equip it with efficient implementation techniques. Of course, it is also important that the implementation is sound with respect to the semantic specification, which is often at a higher level, one that is more amenable for the elaboration of semantic properties.

1.2 Two-level languages

Many applications generate program code: traditional program translators, including compilers for general-purpose and domain specific languages, and source-to-source translators; advanced program optimizers, including partial evaluators and various other semantics-based program transformers; and, relatively more recently, programs that generate code at run-time for efficiency, e.g., Pu et al.'s Synthesis Kernel [91]. Despite this great variety, some essential issues are common to all these applications. For example, we usually represent the code generated with some specific data structure, and we usually expect that the generated code should behave correctly in one way or another.

Two-level languages, and their many recent derivatives (multi-level and multi-stage languages), provide the language abstraction for these applications. Firstly, they support expressing algorithms that generate program code in a specific language, the *object* language, by incorporating types for code representation and associated constructions in the concrete syntax of this language. Often, the code type manifests the type of the generated code, which, through a type-preservation property, captures the common static debugging task of ensuring that the generated code is type correct.

Two-level languages originate as a formalism for describing partial evaluators and compiler back ends. The 1990s saw an increasing interest in using two-level and multi-level languages as the source language for code-generation algorithms. Several interesting multi-level languages have been proposed, with different degree of expressive power. In these studies of multi-level languages, however, the

only general property that has been consistently studied is the type soundness property. Furthermore, the high-level semantics used for these languages are not related to any effective implementation—either such implementation does not exist, or their correctness proof difficult.

The current situation of two-level and multi-level languages, consequently, is somewhat disparate. Their studies divide into two separate camps: on one side, new such languages are designed, but they lead to few applications and scarcely account for the implementation; on the other side, applications often use two-level languages informally to motivate and to describe the algorithms, with a separate effort for their formalization and implementation.

A survey of related work in the area of two-level languages and frameworks is found in Part I, in particular in Chapters 3 and 6.

1.3 Type dependency

Program-generating algorithms, and language-processing algorithms such as interpreters in general, often exhibit a kind of type dependency. That is, the value of some input argument could determine the types of the other input arguments and of the output. The frequent uses of type dependency is not by pure accident. In most cases, these language-processing algorithms treat typed subject programs in the object language, and these object-level types are reflected in the meta-level typing of certain components of the algorithm.

In most cases, we can evade the type dependency by using a universal data type for the type that changes according to the input. But the use of a universal

data type also introduces potential type errors in the form of tag mismatches, which is against the static debugging provided by a static type system. The use of a universal type is particularly unsatisfactory when the end users are required to supply the arguments whose type depends on the rest of the input, as examples in Part II will demonstrate.

Unfortunately, a dependent type system is difficult to implement and use, and is left out by the mainstream programming languages. On the other hand, several languages incorporate certain extensions to the type system that allows a limited form of type dependency, as reviewed in Chapter 12.

1.4 Overview of this dissertation

This work follows the language-based approach to addressing issues in code-generating algorithms.

In Part I, we set up a framework of practical two-level languages for high-level program generation. We start with a simple two-level language for pure call-by-value object languages found in the literature, express several non-trivial applications in this language, and identify and prove the necessary key properties of the two-level languages that support establishing the correctness of these applications themselves. We also establish an effective implementation for the language, by showing a native embedding into a conventional one-level language. From this development, we abstract out the general framework, in particular the key semantic properties. With this general framework in mind, we proceed to design a new, arguably more practical two-level language, which works for call-

by-value object languages with computational effects. We then demonstrate that this new two-level language is an instance of the general framework by establishing its semantic properties, and we apply these properties to applications. The semantics and the proofs are generally more involved, and therefore we keep most of the technical development in the appendix; but the basic structure does not deviate from the general framework.

Due to our use of native-embedding-based implementations instead of the more traditional meta-level implementation, type dependency is more prominent. In Part II, we turn to study a programming technique that supports a limited form of type dependency that is enough to cover a sizable class of practical applications. We formulate these applications with a notion of type-indexed family of values, and develop several approaches to realize type-indexed families in the Hindley-Milner type system, which is the basis for the mainstream functional languages nowadays. These approaches all use type encodings, whose compile-time types reflect the types themselves, which makes the approach type-safe, in the sense that the underlying type system statically guarantees the type dependency. We present both value-dependent type-encodings, which are tied to a specific type-indexed family, and value-independent type-encoding, which are used by various type-indexed families. In particular, for a value-dependent encoding, directly encoding types as the values indexed at the corresponding type is sufficient and practical, while for a value-independent encoding, we use higher-order functors to express the type dependency explicitly, or use a universal type encoding based on embedding and projection functions between universal data types and specific type encodings. Since this part is concerned with a program-

ming technique, we illustrate all the approaches with the Standard ML code of a range of applications.

Building upon the framework of Part I and the programming technique of Part II, in Part III, we derive and implement a more scaled-up application: the second Futamura projection for type-directed partial evaluation, which uses self-application to derive efficient generating extensions (specialized partial evaluators for given programs).[1] Due to the differences between 'traditional', syntax-directed partial evaluation and type-directed partial evaluation, this derivation involves several conceptual and technical steps. These include a suitable formulation of the second Futamura projection and techniques for making TDPE amenable to self-application. We demonstrate this technique with several examples, including compiler generation for Tiny, a prototypical imperative language.

To keep the development of each part more focused, we keep the specific motivations, technical background, related and future work, and concluding remarks separate, leaving them in the relevant parts.

1.5 Themes

While Part I is mainly concerned with a theoretical foundation, Part II with a programming methodology, and Part III with an application, both *correctness* concerns and *implementation* issues are present in all the parts.

Our approach to the soundness of our techniques is semantics-based. We use

[1] This recasts a joint work with Bernd Grobauer. The difference lies in the foundation of the work: whereas the original work is based on Filinski's semantic formulation of TDPE [31], Part III takes advantage of Part I to give a more syntactic formulation.

several different semantic tools to formalize different language abstractions in the paper and to prove them correct. In general, we opt for more "syntactic" semantic tools, such as operational semantics and equational theories, but denotational semantics proves to be handy in certain situations. A survey of these techniques is presented in Section 2.2.

For implementation, instead of the traditional meta-level approach where the language to be implemented is processed as data in the implementation language, we follow an approach called *embedding*, where syntactic phrases of the implemented language are mapped into those of the implementation language. For this embedding approach to be effective and for the mapping of syntactic phrases to be readily usable by the programmer, it is important that the mapping be compositional. This is guaranteed by the notion of *syntactic interpretation* (or *instantiation*), where the syntactic mapping is simply a substitution of base syntactic constituents. A general description of the embedding approach is presented in Section 2.3.

Chapter 2

Preliminaries

2.1 Notational conventions

We employ several notations for taking interpretations of syntax. Specifically, the semantic bracket $[\![\cdot]\!]$ maps a piece of syntax to its semantics, which is a value in a mathematical domain; the syntactic-translation bracket $\{\!|\cdot|\!\}$ maps a piece of syntax to another piece of syntax, probably in a different language. In the special case where the source syntax represents a type while the target syntax represents a program term, we write $\langle\!|\cdot|\!\rangle$ instead, indicating that the syntactic translation is a type encoding.

Because we consider several different languages, we write $\mathsf{L} \vdash J$ to assert a judgment J in the language L, or we write simply J when L is clear from the context. We write $\equiv$ for strict syntactic equality, $\sim_\alpha$ for equality up to α-conversion, and $\triangleq$ for definitions.

Operations (syntactic translations) defined on types τ, say $\{\!|\tau|\!\}$, are ho-

momorphically extended to apply to contexts: $\{\!|x_1 : \tau_1, \ldots, x_n : \tau_n|\!\} \equiv x_1 : \{\!|\tau_1|\!\}, \ldots, x_n : \{\!|\tau_n|\!\}$. A type-preserving translation $\{\!|-|\!\}$ of terms-in-contexts in language L_1 into ones in language L_2 is declared in the form $\boxed{\mathsf{L}_1 \vdash \Delta \rhd E : \sigma} \Longrightarrow \boxed{\mathsf{L}_2 \vdash \{\!|\Delta|\!\} \rhd \{\!|E|\!\} : \{\!|\sigma|\!\}}$.

For a syntactic phrase (a type or a term) p and a substitution Φ (a map from variables to syntactic phrases), we write $p\{\Phi\}$ for the result of applying the substitution Φ to p. We note a substitution in the form $p_1/x_1, \ldots, p_n/x_n$, where each x_i is a variable, and each p_i is the corresponding syntactic phrase.

Meta-variables τ, σ, Γ, and Δ respectively range over two-level types, one-level types, two-level contexts, and one-level contexts. Meta-variables α, β, γ range over type variables.

2.2 Semantics: tools for reasoning

This dissertation uses three kinds of formal semantics in building sound language support: operational semantics, denotational semantics, and axiomatic semantics.

An operational semantics describes the meaning of a program in terms of its operational behavior. Following Landin's work on SECD machine [64], early work of operational semantics mostly took the form of abstract machines. An abstract-machine semantics is very close to an actual implementation, but its lack of abstraction makes high-level reasoning difficult. Later, Plotkin developed structural operational semantics as a more abstract alternative [89]. Structural operational semantics specifies program reductions or evaluations inductively,

thus lending itself to the inductive proof principles.

The specific form of operational semantics we use in this dissertation is evaluation semantics (a.k.a. big-step semantics). In its simplest form, an evaluation semantics presents an inductively defined binary relation between an expression E and a value V, which is a result of evaluating E. Values are special expressions that are canonical in some formal sense.

In an denotational semantics, the meaning of a program is a mathematical object, i.e., its denotation. An important feature of denotational semantics is *compositionality*: the meaning of a syntactic phrase is determined by the meaning of its sub-phrases. Compositionality makes denotational semantics a suitable tool for reasoning about program equivalence: replacing a sub-term in a program with another sub-term does not change the meaning of the whole program, provided that the two sub-terms have the same denotations. Achieving compositionality, on the other hand, often necessitates advanced mathematical constructions. As a well-known example, the attempt to model recursion leads to the development of domain theory [95].

An axiomatic semantics provides an axiomatic system for reasoning about program properties. With an emphasis on its use, axiomatic semantics is the probably the most user-oriented among the three genres of formal semantics. A classical example is the Hoare logic for reasoning about partial correctness of programs, which defines a relation between programs, pre-condition, and post-conditions. For functional programming languages, equational theories are a common form of axiomatic semantics. An equational theory is an axiomatic system for relating terms that are observational equivalent, i.e., those which

are interchangeable in abitrary program context without affecting the program behavior.

Any programming language can be equipped with several different formal semantics. In order to freely reap the benefits of all these semantics, it is important that these semantics agree with each other to some extent. Different levels of agreement exists, but, in practice, the form that is called computational adequacy suffices. Computational adequacy requires that the semantics (in most cases denotational semantics and operational semantics agree) agree on whole programs, but not necessarily on the observable behaviors of arbitrary terms. In general, the observational equivalence induced by operational semantics is the coarsest, while the equivalence from the equational theory is the finest.

In the following we sketch the basic form of a simple functional language, its syntax, its denotational semantics, and its equational theory. Operational semantics vary greatly according to the language, therefore we refrain from giving a general description.

A simple functional language L is given by a pair $(\Sigma, \mathcal{I})$ of a signature Σ and an interpretation $\mathcal{I}$ of this signature. More specifically, the syntax of valid terms and types in this language is determined by Σ, which consists of base type names, and constants with types constructed from the base type names. A set of typing rules generates, from the signature Σ, typing judgments of the form $\Sigma \vdash \Gamma \rhd t : \sigma$ (or $\mathsf{L} \vdash \Gamma \rhd t : \sigma$), which reads "$t$ is a well-formed term of type σ under typing context Γ".

The denotational semantics of types and terms is determined by an interpretation. An interpretation $\mathcal{I}$ of signature Σ assigns domains to base type names,

elements of appropriate domains to literals and constants, and, in the setting of call-by-value languages with effects, also monads to various effects. The interpretation $\mathcal{I}$ extends canonically to the meaning $[\![\sigma]\!]^{\mathcal{I}}$ of every type σ and the meaning $[\![t]\!]^{\mathcal{I}}$ of every term $t : \sigma$ in the language; for a term-in-context $\Gamma \rhd t : \sigma$, its meaning $[\![t]\!]^{\mathcal{I}}$ is a map from Γ to σ in the appropriate category of domains (Kleisli category for call-by-value languages with effects).

The equational theory of the language is generated from a parameterized set of axioms on the constants in Sg via a set of equational rules. Each equation takes the form $\Gamma \rhd t_1 = t_2 : \sigma$. A soundness theorem of the equational theory with respect to the denotational semantics states that all derivable such equations are validated by an interpretation $\mathcal{I}$, i.e., $[\![t_1]\!]^{\mathcal{I}} = [\![t_2]\!]^{\mathcal{I}}$, provided that the axioms on the constants are validated by $\mathcal{I}$.

2.3 An embedding approach to Language implementation

2.3.1 The embedding approach

Assume that we have an implementation of language $\mathsf{L}^{\mathsf{imp}}$, and we want to implement a language L. We can achieve this by writing an L-interpreter in $\mathsf{L}^{\mathsf{imp}}$, or writing an L-compiler in $\mathsf{L}^{\mathsf{imp}}$. A more lightweight approach called embedding might work when $\mathsf{L}^{\mathsf{imp}}$ and L are share most language constructs. In the embedding approach, one creates a library of combinators to deal with those constructs of L that are not immediately present in $\mathsf{L}^{\mathsf{imp}}$. This allows direct embedding of

L-programs in $\mathsf{L}^{\mathsf{imp}}$: the common constructs of L and $\mathsf{L}^{\mathsf{imp}}$ can be left as is, while the extra constructs of L embeds as function invocations to the combinators.

The formal notion of syntactic translation serves to make this process of embedding mathematically precise and reasonable. Concretely, an embedding of L into L is simply a syntactic translation $\{\!|\cdot|\!\}$ from L to $\mathsf{L}^{\mathsf{imp}}$ such that the translation of the common constructs are homomorphic. For example, if both languages are functional, then we should expect that variables, λ-abstractions, and function applications all translate to themselves, i.e., $\{\!|x|\!\} \triangleq x$, $\{\!|\lambda x.t|\!\} \triangleq \lambda x.\{\!|t|\!\}$, and $\{\!|t_1\ t_2|\!\} \triangleq \{\!|t_1|\!\}\,\{\!|t_2|\!\}$. Correctness theorem of the implementation can then be precisely stated. Part I uses the embedding method to implement the two-level languages and prove the implementation correct.

In general, this embedding approach to language implementation applies only when the type systems of the new language L and the implementation language $\mathsf{L}^{\mathsf{imp}}$ are very close, as we cannot plug in a dedicated type inferencer for L. In Part II, however, we shall demonstrate implementing more expressive type systems using the embedding approach.

2.3.2 A special form of embedding: instantiation

It is helpful to understand the precise concept of embedding for the case when both $\mathsf{L}^{\mathsf{imp}}$ and L are simple functional languages of the form described in Section 2.2. In this case, the only "extra" constructs in L are the types and constants in its signature. Therefore, we only need to map them into the $\mathsf{L}^{\mathsf{imp}}$-phrases.

In fact, the syntactic counterpart of the notion of an interpretation is that of an *instantiation* (also known as a *syntactic interpretation*), which composi-

tionally maps syntactic phrases in a language L to syntactic phrases in (usually) another language L'. The following definition of instantiations uses the notion of substitution. For a substitution Φ, we write $t\{\Phi\}$ and $\sigma\{\Phi\}$ to denote the application of Φ to term t and type σ, respectively.

Definition 2.1 (Instantiation). *Let L and L' be two languages with signatures Σ and Σ', respectively. An instantiation Φ of Σ-phrases (terms and types) into language L' is a substitution that maps the base types in Σ to Σ'-types, and maps constants $c : \sigma$ to closed Σ'-terms of type $\sigma\{\Phi\}$.*

We also refer to the term $t\{\Phi\}$ as the instantiation of the term t under Φ, and the type $\sigma\{\Phi\}$ as the instantiation of the type σ under Φ.

It should be obvious that an interpretation of a language L' and an instantiation of a language L in language L' together determine an interpretation of L.

2.4 Type-directed partial evaluation: an informal introduction

2.4.1 Partial evaluation

Given a general program $p : \sigma_S \times \sigma_D \to \sigma_R$ and a fixed *static* input $s : \sigma_S$, partial evaluation (a.k.a. program specialization) yields a specialized program $p_s : \sigma_D \to \sigma_R$. When this specialized program p_s is applied to an arbitrary *dynamic* input $d : \sigma_D$, it produces the same result as the original program applied to the complete input (s, d), i.e., $[\![p_s d]\!] = [\![p(s, d)]\!]$ (Here, $[\![\cdot]\!]$ maps a piece of program text to

its denotation.) Often, some computation in program p can be carried out independently of the dynamic input d, and hence the specialized program p_s is more efficient than the general program p. In general, specialization is carried out by performing the computation in the source program p that depends only on the static input s, and generating program code for the remaining computation (called residual code).

We will present further details on the area of partial evaluation in Part III of the dissertation, where it is the subject of study.

Partial evaluation through normalization by evaluation In a suitable setting, partial evaluation can be carried out by normalization. Consider, for example, the pure simply typed λ-calculus, in which computation means β-reduction. Given two λ-terms $p\colon \tau_1 \to \tau_2$ and $s\colon \tau_1$, bringing the application ps into β-normal form specializes p with respect to s. For example, normalizing the application of the K-combinator $K = \lambda x.\lambda y.x$ to itself yields $\lambda y.\lambda x.\lambda y'.x$.

Type-directed partial evaluation (TDPE), due to Danvy [12], builds on this idea of specialization by normalization, in a somewhat unconventional fashion. The distinct flavor of TDPE lies in the normalization algorithm, which builds upon an evaluator (interpreter or compiler) and is free of expensive symbol reductions. On pure call-by-name λ-calculus, TDPE coincides with Berger and Schwichtenberg's *Normalization by Evaluation* (NbE) [6, 17]. In fact, in Part I and Part II, we will only concern ourselves with (the formulation, implementation, and correctness proofs of) the normalization algorithm of TDPE, and refer to it simply as TDPE; subsequently, the focused development on TDPE itself

and of self-application techniques for TDPE appear inPart III Roughly speaking, NbE works by extracting the normal form of a term from its meaning, where the extraction function is coded in the object language.

Example 2.2. *Let* L *be a higher-order functional language in which we can define a type* exp *of term representations. Consider the combinator* $\mathsf{K} = \lambda x.\lambda y.x$*—the term* KK *is of type* $\forall\alpha.\alpha\to\alpha\to\alpha\to\alpha$*, which can be instantiated to the monotype* $\mathsf{exp}\to\mathsf{exp}\to\mathsf{exp}\to\mathsf{exp}$*. We want to extract a* β*-normal form from its meaning.*

Since $\mathsf{exp}\to\mathsf{exp}\to\mathsf{exp}\to\mathsf{exp}$ *is the type of a function that takes three arguments, one can infer that a (fully-*η*-expanded)* β*-normal form of* KK *must be of the form* $\underline{\lambda}v_1.\underline{\lambda}v_2.\underline{\lambda}v_3.\mathsf{t}$ *(we underline term representations to distinguish them from terms), for some term* $\mathsf{t} : \mathsf{exp}$*. Intuitively, the only natural way to generate the term* t *from the meaning of term* KK *is to apply it to the representations of the terms* v_1*,* v_2 *and* v_3*. The result of this application is* v_2*. Thus, we can extract the normal form of* KK *as* $\underline{\lambda}v_1.\underline{\lambda}v_2.\underline{\lambda}v_3.v_2$*.*

2.4.2 Pure TDPE in ML

In this section, we illustrate TDPE for an effect-free fragment of ML without recursion, which we call *Pure TDPE*. For this fragment, the call-by-name and call-by-value semantics agree, which allows us to directly use Berger and Schwichtenberg's NbE for call-by-name λ-calculus as the core algorithm (recall that ML is a call-by-value functional language).

NbE works by extracting the normal form of a λ-term from its meaning, by regarding the term as a higher-order code-manipulation function. The extrac-

tion functions are type-indexed coercion functions coded in the implementation language. To carry out partial evaluation based on NbE, TDPE thus needs to use a code-manipulating version of the subject λ-term. Such a λ-term, in general, could contain constant functions that cannot be statically evaluated; these constants have to be replaced with code-manipulating functions.

Pure simply-typed λ-terms We first consider TDPE only for pure simply-typed λ-terms. We use the type exp in Figure 2.1 to represent code (as it is used in Example 2.2). To use ML with exp as an informal two-level language for describing code-generation algorithms, in the following we will write $\underline{\lambda}x.E$ as shorthand for `LAM(`x`, `E`)`, $E_1 \underline{@} E_2$ as shorthand for `APP(`E_1`,`E_2`)`, and an occurrence of a $\underline{\lambda}$-bound variable x as shorthand for `VAR(`x`)`. Following the convention of the λ-calculus, we use $\underline{@}$ as a left-associative infix operator.

```
datatype Exp = VAR of string
             | LAM of string * Exp
             | APP of Exp * Exp
```

Figure 2.1: A data type for representing terms

Let us for now only consider ML functions that correspond to pure λ-terms with type τ of the form $\tau ::= \bullet \mid \tau_1 \to \tau_2$, where '$\bullet$' denotes a base type. The TDPE algorithm takes as input terms of the types $\overline{\tau} = \tau\{\mathsf{exp}/\bullet\}$; that is, a value representing either code (when $\overline{\tau} = \mathsf{exp}$), or a code-manipulating function (at higher types).

Figure 2.2 shows the TDPE algorithm: For every type τ, we define inductively a pair of functions $\downarrow^\tau : \overline{\tau} \to \mathsf{exp}$ (reification) and $\uparrow_\tau : \mathsf{exp} \to \overline{\tau}$ (reflection).

Reification is the function that extracts a normal form from the value of a code-manipulation function, using reflection as an auxiliary function. We explain reification and reflection through the following examples.

$$
\begin{array}{rcll}
\downarrow^{\bullet} e & = & e & \\
\downarrow^{\tau_1 \to \tau_2} f & = & \underline{\lambda} x.\downarrow^{\tau_2} (f(\uparrow_{\tau_1} x)) & (x \text{ is fresh}) \\
 & & & \\
\uparrow_{\bullet} e & = & e & \\
\uparrow_{\tau_1 \to \tau_2} e & = & \lambda x.\uparrow_{\tau_2} (e \underline{@} (\downarrow^{\tau_1} x)) &
\end{array}
$$

Figure 2.2: Reification and reflection

Example 2.3. *We revisit the normalization of* KK *from Example 2.2 (page 17). For the type* $\bullet \to \bullet \to \bullet \to \bullet$ *the equations given in Figure 2.2 define reification as*

$$\downarrow^{\bullet \to \bullet \to \bullet \to \bullet} e = \underline{\lambda} x.\underline{\lambda} y.\underline{\lambda} z.exyz.$$

For every argument of base type '$\bullet$*', a lambda-abstraction with a fresh variable name is created. Given a code-manipulation function of the type* $\mathsf{exp} \to \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$*, a code representation of the body is then generated by applying this function to the code representations of the three bound variables. Evaluating* $\downarrow^{\bullet \to \bullet \to \bullet \to \bullet} (\mathsf{KK})$ *yields* $\underline{\lambda} x.\underline{\lambda} y.\underline{\lambda} z.y$.

What happens if we want to extract the normal form of $t : \tau_1 \to \tau_2$ where τ_1 is not a base type? The meaning of t cannot be directly applied to the code representing a variable, since the types do not match: $\overline{\tau_1} \neq \mathsf{exp}$. This is where

the *reflection* function $\uparrow_\tau : \mathsf{exp} \to \overline{\tau}$ comes in; it converts a code representation into a code-generation function:

Example 2.4. *Consider* $\tau_1 = \bullet \to \bullet \to \bullet \to \bullet$*:*

$$\uparrow_{\bullet\to\bullet\to\bullet\to\bullet} e = \lambda x.\lambda y.\lambda z.e\underline{@}x\underline{@}y\underline{@}z$$

For any term representation e*,* $\uparrow_{\bullet\to\bullet\to\bullet\to\bullet} e$ *is a function that takes three term representations and constructs a representation of the application of* e *to these term representations. It is used, e.g., when reifying the term* $\lambda x.\lambda y.xyyy$ *with* $\downarrow^{(\bullet\to\bullet\to\bullet\to\bullet)\to\bullet\to\bullet}$*.*

Adding constants So far we have seen that we can normalize a pure simply-typed λ-term by (1) coding it in ML, interpreting all the base types as type exp, so that its value is a code-manipulation function, and (2) applying reification at the appropriate type. Treating terms with constants follows the same steps, but the situation is slightly more complicated. Consider, for example, the ML expression $\lambda z.$`mult` 3.14 z of type `real` $\to$ `real`, where `mult` is a curried version of multiplication over reals. This function cannot be used as a code-manipulation function, since its type is not constructed from the type exp. The solution is to use a non-standard, code-generation version $\texttt{mult}_{\mathsf{r}} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ of `mult`. We also *lift* the constant 3.14 into exp using a lifting-function $\mathsf{LIT}_{\texttt{real}} : \texttt{real} \to \mathsf{exp}$. (This operation requires a straightforward extension of the data type exp with an additional constructor `LIT_REAL`.) Reflection can then be used to construct a code-generation version $\texttt{mult}_{\mathsf{r}}$ of `mult`:

Example 2.5. *A code-generation version* $\mathtt{mult_r} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{exp}$ *of* $\mathtt{mult} : \mathtt{real} \to \mathtt{real} \to \mathtt{real}$ *is given by*

$$\mathtt{mult_r} \;=\; \uparrow_{\bullet\to\bullet\to\bullet} \text{“}\mathit{mult}\text{”} \;=\; \lambda x.\lambda y.\text{“}\mathit{mult}\text{”}\underline{@}x\underline{@}y,$$

where “*mult*” *(more accurately,* $\mathtt{VAR}$ “*mult*”*) is the code representation of a constant with name* ***mult****. Now applying the reification function* $\downarrow^{\bullet\to\bullet}$ *to the term*

$$\lambda z.(\mathtt{mult_r}\ (\mathsf{LIT}_{\mathtt{real}}\ 3.14)\ z)$$

evaluates to $\underline{\lambda}x.\text{“}\mathit{mult}\text{”}\underline{@}3.14\underline{@}x$.

Towards partial evaluation: binding-time annotations In the framework of TDPE, the partial evaluation of a (curried) program $p : \sigma_S \to \sigma_D \to \sigma_R$ with respect to a static input $s : \sigma_S$ is carried out by normalizing the application ps. We could use a code-generation version for all the constants in this term; reifying the meaning will carry out all the β-reductions, but leave all the constants in the residual program—no static computation involving constants is carried out. However, this is not good enough: One would expect that the application ps enables also computations involving constants, not only β-reductions. Partial evaluation, of course, should also carry out such computations. This is achieved by leaving the involved constants as is; or, in other words, instantiating these constants to themselves.

In general, to perform TDPE for a term, one needs to decide for each occurrence of a constant, whether to use the original constant or a code-generation instantiation of it; appropriate lifting functions have to be inserted where necessary. The result must type-check, and its partial application to the static input

must represent a code-manipulation function (i.e., its type is built up from only the base type exp), so that we can apply the reification function.

This process of classification corresponds to a binding-time annotation phase, as will be made precise in the framework of a two-level language (Sections 4.5, 5.3, and 15.1) Basically, a source term is turned into a well-formed two-level term by marking constants as static or dynamic, and inserting lifting functions where needed. In general, one tries to reduce the number of occurrences of dynamic constants in term t, so that more static computation involving constants is carried out during partial evaluation. Because only constant occurrences have to be annotated, this can, in practice, be done by hand. Given an annotated term t_{ann}, we call the corresponding code-manipulation function its *residualizing instantiation* $\overline{t_{\mathsf{ann}}}$. It arises from t_{ann} by instantiating each dynamic constant c with its code-generation version c_{r}, each static constant with itself, and each lifting function with the appropriate coercion function into exp. If t is of type σ, then its normal form can be calculated by reifying $\overline{t_{\mathsf{ann}}}$ at type σ (remember that reification only distinguishes a type's shape—all base types are treated equally as '$\bullet$'):

$$\mathsf{NF}(t) = [\![\downarrow^{\sigma} \overline{t_{\mathsf{ann}}}]\!];$$

Recall that $[\![\cdot]\!]$ maps a piece of code to its denotation; or, informally, it delivers the result of "executing" the program.

Partial evaluation of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ with respect to a static input $s : \sigma_S$ thus proceeds as follows:

- binding-time annotate p and s as p_{ann} and s_{ann}, respectively;[1] the term

[1] That the static input also needs to be binding-time annotated may at first seem strange. This is

$\lambda x.\overline{p_{\mathsf{ann}}}(\overline{s_{\mathsf{ann}}}, x)$ must be a code-manipulation function of type $\overline{\sigma_D \to \sigma_R}$ (recall that $\overline{\tau}$ arises from τ by instantiating each base type with exp).

- carry out partial evaluation by reifying the above term at type $\sigma_D \to \sigma_R$:

$$p_s = [\![\downarrow^{\sigma_D \to \sigma_R} \lambda x.\overline{p_{\mathsf{ann}}}(\overline{s_{\mathsf{ann}}}, x)]\!]$$

Example 2.6. *Consider the function*

$$\texttt{height} = \lambda(a : \texttt{real}).\lambda(z : \texttt{real}).\texttt{mult}\ (\texttt{sin}\ a)\ z.$$

Suppose we want to specialize height *to a static input* a : real*. It is easy to see that the computation of* sin *can be carried out statically, but the computation of* mult *cannot—*mult *is a dynamic constant. This analysis results in a two-level term* $\texttt{height}_{\mathsf{ann}}$*, in which* sin *is marked as static,* mult *as dynamic, and a lifting function has been inserted to turn the static application of* sin *to a into dynamic. The residualizing instantiation of* $\texttt{height}_{\mathsf{ann}}$ *instantiates* sin *with the standard sine function, the lifting function with a coercion function from* real *into* exp*, and* mult *with a code-generation version as introduced in Example 2.5 (page 20):*

$$\overline{\texttt{height}_{\mathsf{ann}}} = \lambda(a : \texttt{real}).\lambda(z : \texttt{Exp}).\texttt{mult}_{\mathsf{r}}\ (\mathsf{LIT}_{\texttt{real}}(\texttt{sin}\ a))\ z$$

Now $(\overline{\texttt{height}_{\mathsf{ann}}}\ \frac{\pi}{6})$ *has type* Exp → Exp*, i.e., it is a code-manipulation function. Thus, we can specialize* height *with respect to* $\frac{\pi}{6}$ *by evaluating* $\downarrow^{\bullet\to\bullet}(\overline{\texttt{height}_{\mathsf{ann}}}\ \frac{\pi}{6})$*, which yields* $\underline{\lambda}x.\text{“}\mathsf{mult}\text{”}\underline{@}0.5\underline{@}x$

natural, however, because TDPE also accepts higher-order values as static input. For a static input of base type, the binding-time annotation is trivial.

Notice that instantiation in a binding-time annotated term t_{ann} of every constant with itself and of every lifting function with the identity function yields a term $|t_{\mathsf{ann}}|$ that has the same denotation as the original term t; we call $|t_{\mathsf{ann}}|$ the *standard* (also *evaluating*) *instantiation* of t_{ann}.

Part I

A framework for two-level languages

Chapter 3

Introduction to Part I

3.1 Background

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

Two-level languages provide intuitive notations for writing code-generation programs succinctly. They incorporate both static constructs and dynamic constructs for modeling the binding-time separation. Their design usually considers certain semantic aspects of the object languages. For example, the typing safety

of a two-level language states not only that (static) evaluation of well-typed programs does not go wrong, but also that the generated code is well-typed in the object language. The semantic benefit, however, often comes at the price of implementation efficiency and its related correctness proof.

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

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

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

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

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

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

If we are working in a standard, high-level operational semantics that describes evaluation as symbolic computations on the two-level terms, then the solution to the name-capturing problem is simple: Dynamic λ-bound variables, like usual bound variables, should be subject to renaming during a non-capturing substitution $E\{E'/x\}$ (which is used in the evaluation of static applications). Therefore, in the earlier example, the two-level term $\underline{\lambda}x.(\lambda f.\underline{\lambda}x.f\underline{@}x)x$ does not evaluate to $\underline{\lambda}x.\underline{\lambda}x.x\underline{@}x$, but to $\underline{\lambda}x.\underline{\lambda}y.x\underline{@}y$. This precise issue is referred to as "hygienic macro expansion" in Kohlbecker's work [62, 63].

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. [57, 58, 59]: Serving as an intermediate language of offline partial evaluators, a two-level language is the staged version of a corresponding one-level language. In this context, in addition to typing safety, another property, which we call *annotation erasure*, is important for showing the correctness of partial evaluators: The result of two-level evaluation has the same semantics as the unstaged version of the program. Taking up the earlier example again, we can see that the unstaged version of $\underline{\lambda}x.(\lambda f.\underline{\lambda}x.f\underline{@}x)x$, i.e., $\lambda x.(\lambda f.\lambda x.f\,x)x$, is β-equivalent to the generated term $\lambda x.\lambda y.x\,y$. In the symbolic framework, it is relatively easy to establish annotation erasure, at least in a call-by-name, effect-free setting.

For realistic implementations of two-level languages, capture-avoiding substitution is expensive and undesirable. Indeed, most implementations use some strategy to generate variables such that they do not conflict with each other. Unsurprisingly, it is more difficult to reason about these implementations. In

fact, existing work that proved annotation erasure while taking the name generation into account used denotational-semantics formulations and stayed clear of operational semantics [31, 37, 75] (see Section 6.2 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 [26] and of modal logic [27]. Their work fostered the further development of multi-level languages such as MetaML [76]. 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 [18], which is an optimization of Plotkin's original CPS transformation [87]. 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 [12]. 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 [31, 32]. On the other hand, the intuitive connection between TDPE and conventional two-level languages has not been formalized.

3.2 This work

The specific thesis of Part I 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-

[1] Without constants, the call-by-name version of TDPE coincides with Berger and Schwichtenberg's notion of normalization by evaluation [6].

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 Part I The remainder of Part I fleshes out the preceding ideas with two instances of the framework. The first is a canonical two-level language nPCF^2 for a call-by-name object language (call-by-**n**ame "PCF of two-level languages", following Moggi [75]). The second, designed from scratch while taking the aforementioned properties (in particular, annotation erasure and native implementability) into account, is a more practically relevant two-level language vPCF^2: one with an instance of Moggi's call-by-**v**alue computational λ-calculus as its object language.

In Chapter 4 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. At the end of this case study we abstract out, in Chapter 4.7, our general framework, in particular the desired properties and the

corresponding proof obligations they support. With this framework in mind, in Chapter 5 we design vPCF^2, prove its properties, and apply them to the example of call-by-value TDPE. We present the related work in Chapter 6 and conclude in Chapter 7. The detailed proofs and development are given in the appendices.

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

Chapter 4

The call-by-name two-level language nPCF²

We present a canonical call-by-name (CBN) two-level language nPCF² (Section 4.1), cast the example of a one-pass CPS transformation as an nPCF² program (Section 4.2), and use an erasure argument to prove its correctness (Section 4.3). Building on a native embedding of nPCF² into a conventional language (Section 4.4), we formulate CBN TDPE in nPCF² and show its semantic correctness as well as its syntactic correctness (Sections 4.5 and 4.6).

4.1 Syntax and semantics

For the various languages in Part I, we fix a set of base syntactic constituents (Figure 4.1). Figure 4.2 shows the type system ($\Gamma \rhd E : \tau$) 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 $\mathsf{if}_\sigma : \mathsf{bool} \to \sigma \to \sigma$ in Sg.

In addition to the conventional CBN static part,[1] the language nPCF^2 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 b, called the literals, can be "lifted" into the code types $\bigcirc\mathsf{b}$ with $\$_\mathsf{b}$, so that the result of static evaluation can appear in the generated code. The dynamic constructs are akin to data constructors of the familiar algebraic types, but with the notable exception that the dynamic λ-abstraction is a binding operator: As mentioned in the introduction, the variables introduced are, like usual bound variables, subject to renaming during a non-capturing substitution $E\{E'/x\}$ (which is used in the evaluation of static applications).

The evaluation judgment of the form $E \Downarrow V$ reads that evaluation of the term E leads to a *value* V. Evaluation is deterministic modulo α-conversion:

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

Base types $\mathsf{b} \in \mathbb{B}$: bool (boolean type), int (integer type)

Literals ℓ: $\mathbb{L}(\mathsf{bool}) = \{\mathsf{tt}, \mathsf{ff}\}, \mathbb{L}(\mathsf{int}) = \{\ldots, -1, 0, 1, \ldots\}$

Binary operators $\otimes$: $+, - : \mathsf{int} \times \mathsf{int} \to \mathsf{int}$,

$=, < : \mathsf{int} \times \mathsf{int} \to \mathsf{bool}$

Figure 4.1: Base syntactic constituents

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

b. **Syntax**

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

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

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

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

$\mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid \underline{d} \mid \underline{\lambda} x.E \mid E_1 \underline{@} E_2$

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

(Static)

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

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

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

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

(Dynamic)

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

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

Figure 4.2: The two-level call-by-name language nPCF^2 (1/2)

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

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

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

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

(Static) $[\mathit{lit}]\ \dfrac{}{\ell \Downarrow \ell}$ $\quad [\mathit{lam}]\ \dfrac{}{\lambda x.E \Downarrow \lambda x.E}$ $\quad [\mathit{app}]\ \dfrac{E_1 \Downarrow \lambda x.E' \quad E'\{E_2/x\} \Downarrow V}{E_1\, E_2 \Downarrow V}$

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

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

(Dynamic) $[\underline{\mathit{lift}}]\ \dfrac{E \Downarrow \ell}{\$_{\mathsf{b}}E \Downarrow \$_{\mathsf{b}}\ell}$ $\quad [\underline{\mathit{var}}]\ \dfrac{}{x \Downarrow x}$ $\quad [\underline{\mathit{cst}}]\ \dfrac{}{\underline{d} \Downarrow \underline{d}}$

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

Figure 4.2: The two-level call-by-name language nPCF^2 (2/2)

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

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

$\mid \mathbf{fix}\, E \mid \mathbf{if}\ E_1\ E_2\ E_3 \mid E_1 \otimes E_2$

Contexts $\Delta ::= \cdot \mid \Delta, x : \sigma$

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

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

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

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

Figure 4.3: The one-level call-by-name language nPCF

annotations (erasure will be made precise in Section 4.3).

Because evaluation proceeds under dynamic λ-abstractions, intermediate results produced during the evaluation can contain free dynamic variables [76]. 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 4.1 (Type preservation). *If* $\bigcirc\Delta \rhd E : \tau$ *and* $E \Downarrow V$*, then* $\bigcirc\Delta \rhd V : \tau$*. (*$\bigcirc\Delta$ *is the element-wise application of the* $\bigcirc(-)$ *constructor to the context* Δ*.)*

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

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

Figure 4.3 shows the corresponding one-level language nPCF. The language includes not only the constructs of the object language, but also the static constructs of nPCF^2. Though the static constructs will not appear in the generated code, they are needed to specify and prove the semantic correctness of the generated code.

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

4.2 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 4.4a) into continuation-passing style (CPS).[2] The typed formulation [69] of Plotkin's original transformation [87] maps a term E directly into a one-level term $\{\!|E|\!\}_{\mathsf{p}\kappa}$ (Figure 4.4b), but it generates a lot of administrative redexes—roughly all the bindings named k introduce an extra redex—and to remove these redexes requires a separate pass. Danvy and Filinski's one-pass CPS transformation instead maps the term into a two-level program $\{\!|E|\!\}_{\mathsf{df}\kappa}$ (Figure 4.4c); evaluating $\{\!|E|\!\}_{\mathsf{df}\kappa}$ produces the resulting CPS term. The potential administrative redexes are annotated as static, and thus are reduced during the evaluation of $\{\!|E|\!\}_{\mathsf{df}\kappa}$. Intuitively, the one-pass transformation is derived by staging the program $\{\!|E|\!\}_{\mathsf{p}\kappa}$ [18].

By the definition of the translation, the two-level program $\{\!|E|\!\}_{\mathsf{df}\kappa}$ does not use the fixed-point operator. We can prove that the evaluation of such a term always terminates using a standard logical-relation argument (note that, with respect to the termination property, the code type behaves the same as a usual base type like int).[3] The question is how to ensure that the resulting term has the same behavior as the output of Plotkin's original transformation, $\{\!|E|\!\}_{\mathsf{p}\kappa}$. An intuitive argument is that erasing the annotations in $\{\!|E|\!\}_{\mathsf{df}\kappa}$ produces a term which is $\beta\eta$-equivalent to $\{\!|E|\!\}_{\mathsf{p}\kappa}$.

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

[3] Less directly, we can also use the embedding translation introduced in Section 4.4 and its associated correctness theorem: The embedding of a term without fixed-point operators does not use the fixed-point operator either, and thus its evaluation terminates in the standard operational semantics.

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

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

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

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

b. Plotkin's original transformation:

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

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

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

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

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

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

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

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

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

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

Terms:

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

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

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

The complete translation

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

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

Figure 4.4: Call-by-value CPS transformation

4.3 Semantic correctness of the generated code: erasure

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

Definition 4.2 (Erasure). *The (annotation) erasure of a* nPCF^2*-phrase is the* nPCF*-phrase given as follows.*

Types: $|\bigcirc\sigma| = \sigma$, $|\mathsf{b}| = \mathsf{b}$, $|\tau_1 \to \tau_2| = |\tau_1| \to |\tau_2|$.

Terms: $|x| = x$, $|\$_{\mathsf{b}} E| = E$, $|\underline{d}| = d$, $|\underline{\lambda} x.E| = \lambda x.|E|$,

$$|E_1 \underline{@} E_2| = |E_1|\,|E_2|.$$

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

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

Theorem 4.3 (Annotation erasure). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF} \vdash \Delta \rhd |E| = |V| : |\tau|$.

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

With Theorem 4.3, 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 4.4 (Correctness of one-pass CPS). *If* $\mathsf{v}\Lambda \vdash \Delta \rhd E : \sigma$ *and* $\mathsf{nPCF}^2 \vdash \{\!|E|\!\}_{\mathsf{df}\kappa} \Downarrow \mathcal{O}$ *then* $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\mathcal{O}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$.

Proof. A simple induction on E establishes that $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\{\!|E|\!\}_{\mathsf{df}^2\kappa}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$, which has the immediate corollary that $\mathsf{nPCF} \vdash \{\!|\Delta|\!\}_{\mathsf{p}\kappa} \rhd |\{\!|E|\!\}_{\mathsf{df}\kappa}| = \{\!|E|\!\}_{\mathsf{p}\kappa} : K\{\!|\sigma|\!\}_{\mathsf{p}\kappa}$. We then apply Theorem 4.3. □

The proof of Proposition 4.4 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 4.3 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 4.3 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.

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

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

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

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

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

The language nPCF^{Λ} has a standard, domain-theoretical CBN denotational semantics [73],[4] which interprets the types as follows:

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

where $\mathbf{Z}$, $\mathbf{B}$ and $\mathbf{E}$ are respectively the set of integers, the set of booleans, and the set of raw terms (i.e., the inductive set given as the smallest solution to the equation $X = \mathbf{Z}+\mathbf{Z}+\mathbf{B}+\mathbf{Cst}+\mathbf{Z}\times X+X\times X$). Without giving the detailed semantics

[4] This is different from a lazy CBN semantics [107], which models Haskell-like languages where higher-order types are observable; there, function spaces are lifted, and the η-rule does not hold.

(which can be found in Appendix B), we remark that (1) the equational theory is sound with respect to the denotational semantics: If $\mathsf{nPCF}^{\Lambda} \vdash \Delta \rhd E_1 = E_2 : \sigma$, then $[\![E_1]\!] = [\![E_2]\!]$, and (2) the evaluation function for closed terms of base types induced from the denotational semantics has (by its computational adequacy with respect to a environment-based (i.e., not substitution-based) call-by-name evaluation semantics where evaluation of the data constructors are strict; the proof of adequacy adapts the standard proof [88]) efficient implementations that do not perform capture-avoiding substitutions.

Definition 4.5 (Embedding of nPCF^{2} into nPCF^{Λ}: $\{\!|-|\!\}_{\mathsf{n}\epsilon}$).

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

where we use the following terms:

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

Static constructs are translated homomorphically.

The three terms used in the embedding translation are nPCF^{Λ}-terms themselves, and are kept as-is in the result of the translation. For instance, $\{\!|f \underline{@} x|\!\}_{\mathsf{n}\epsilon}$ is the term $\underline{@}^{\mathsf{n}\epsilon} f x \equiv (\lambda m.\lambda n.\lambda i.\mathsf{APP}(mi, ni)) f x$, not the simplified term $\lambda i.fi, xi$. This is crucial for the validity of the following substitution lemma (Lemma 4.6); moreover, this also models the actual implementation, where the dynamic con-

structs are provided as combinators in the implementation language nPCF^{Λ}.

The translation uses a de Bruijn-level encoding for generating variable bindings. Furthermore, a dynamic λ-abstraction is translated using a static λ-abstraction[5] and thus the two terms have the same binding behavior—a fact reflected in the following substitution lemma.

Lemma 4.6 (Substitution lemma for $\{\!|-|\!\}_{\mathsf{n}\epsilon}$). *If* $\mathsf{nPCF}^{2} \vdash \Gamma, x : \tau' \rhd E : \tau$ *and* $\mathsf{nPCF}^{2} \vdash \Gamma \rhd E' : \tau'$, *then* $\{\!|E\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \sim_{\alpha} \{\!|E|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\}$.

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

Lemma 4.7 (Evaluation preserves translation). *If* $\mathsf{nPCF}^{2} \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^{2} \vdash E \Downarrow V$, *then* $\mathsf{nPCF}^{\Lambda} \vdash \{\!|\bigcirc\Delta|\!\}_{\mathsf{n}\epsilon} \rhd \{\!|E|\!\}_{\mathsf{n}\epsilon} = \{\!|V|\!\}_{\mathsf{n}\epsilon} : \{\!|\tau|\!\}_{\mathsf{n}\epsilon}$.

Proof. By induction on $\mathsf{nPCF}^{2} \vdash E \Downarrow V$. For $E \equiv E_1\, E_2$, we use Lemma 4.6. □

Lemma 4.8 (Translation of code-typed value). *If* $\mathsf{nPCF}^{2} \vdash v_1 : \bigcirc\sigma_1, \ldots, v_n : \bigcirc\sigma_n \rhd \mathcal{O} : \bigcirc\sigma$, *then there is a value* $t : \Lambda$ *such that*

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

[5] This is an instance of higher-order abstract syntax [86]. 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.

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

Lemma 4.9 (Computational adequacy). *If* $\mathsf{nPCF}^2 \vdash \triangleright E : \bigcirc\sigma$, *and there is a* nPCF^Λ*-value* $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$, *then* $\exists\mathcal{O}.E \Downarrow \mathcal{O}$.

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

Theorem 4.10 (Correctness of embedding). *If* $\mathsf{nPCF}^2 \vdash \triangleright E : \bigcirc\sigma$, *then the following statements are equivalent.*

(a) There is a value $\mathcal{O} : \bigcirc\sigma$ *such that* $\mathsf{nPCF}^2 \vdash E \Downarrow \mathcal{O}$.

(b) There is a value $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$.

When these statements hold, we further have that

(c) $\mathsf{nPCF} \vdash \triangleright\mathcal{D}(t) : \sigma$ *and* $|\mathcal{O}| \sim_\alpha \mathcal{D}(t)$.

Proof. (a) ⇒ (b),(c): We combine Lemmas 4.7 and 4.8 to show the existence of value $t : \Lambda$ such that (c) holds and $\mathsf{nPCF}^\Lambda \vdash \triangleright\{\!|E|\!\}_{\mathsf{n}\epsilon}(1) = t : \Lambda$, which implies (b) by the soundness of the type theory.

(b) ⇒ (a),(c): By Lemma 4.9, we have (a); by (a) ⇒ (c), we have a value $t' : \Lambda$ that satisfies (c). It is easy to show that $t \equiv t'$. □

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.

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

We now turn to a bigger example: Type-Directed Partial Evaluation (TDPE) [15]. Following Filinski's formalization [31], we describe TDPE as a native normalization process for fully dynamic terms (i.e., terms whose types are built solely from dynamic base types) in the somewhat different two-level language $\mathsf{nPCF}^{\mathsf{tdpe}}$. Here, by a native normalization process, we mean an normalization algorithm that is implemented through a native embedding from $\mathsf{nPCF}^{\mathsf{tdpe}}$ into the implementation language.

The syntax of $\mathsf{nPCF}^{\mathsf{tdpe}}$ is displayed in Figure 4.5a. The language $\mathsf{nPCF}^{\mathsf{tdpe}}$ differs from nPCF^{2} in that only base types are binding-time annotated as static (b) or dynamic (b^{D}, instead of $\bigcirc\mathsf{b}$, for clarity), and the language does *not* have any dynamic type constructors (like the dynamic function type in nPCF^{2}). Apart from lifted literals, dynamic constants d^{D} are the only form of term construction that introduces dynamic types. Their types, written in the form σ^{D}, are the

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

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

Raw terms $\quad E \ ::= \ x \mid \ell \mid \lambda x.E \mid E_1\, E_2 \mid \mathbf{fix}\, E_1$

$\qquad\qquad \mid \mathbf{if}\, E_1\, E_2\, E_3 \mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid d^\partial$

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

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

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

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

b. Standard instantiation (TDPE-erasure)

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

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

Figure 4.5: Call-by-name type-directed partial evaluation (1/2)

fully dynamic counterpart of the constants' type σ in the object language: For example, for the object-level constant $\mathsf{eq} : \mathsf{int} \to \mathsf{int} \to \mathsf{bool}$, the corresponding dynamic constant is $\mathsf{eq}^\partial : \mathsf{int}^\partial \to \mathsf{int}^\partial \to \mathsf{bool}^\partial$; consequently, for what we write $\underline{\mathsf{eq}}\underline{@}(\$_{\mathsf{int}}(1+2)) : \bigcirc(\mathsf{int} \to \mathsf{bool})$ in nPCF^2, we write $\mathsf{eq}^\partial\,(\$_{\mathsf{int}}(1+2)) : \mathsf{int}^\partial \to \mathsf{bool}^\partial$ in $\mathsf{nPCF}^{\mathsf{tdpe}}$. Let us stress that there is no binding-time annotation for applications here.

The semantics is described through a *standard instantiation*[6] into the one-

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

c. Extraction functions $\Downarrow^\sigma$ and $\Uparrow_\sigma$:

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

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

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

d. Residualizing instantiation

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

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

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

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

Figure 4.5: Call-by-name type-directed partial evaluation nPCF^2 (2/2)

level language nPCF (Figure 4.5b), which amounts to erasing all the annotations; thus we overload the notation of erasure here.

Normalizing a closed $\mathsf{nPCF}^{\mathsf{tdpe}}$-term E of fully dynamic type $\sigma^{\mathfrak{d}}$ amounts to finding a nPCF-term $E' : \sigma$ in long $\beta\eta$-normal form (fully η-expanded terms

with no β-redexes; see Section 4.6 for detail) such that $\mathsf{nPCF} \vdash \rhd |E| = E' : \sigma$. For example, normalizing the term $\mathsf{eq}^{\mathfrak{d}}\,(\$_{\mathsf{int}}(1+2))$ should produce the object term $\lambda x.\mathsf{eq}\,3\,x$. As in Filinski's treatment, this notion of normalization leaves the dynamic constants uninterpreted—E and E' need to be the same under all interpretations of constants, since there are no equations for dynamic constants.

The TDPE algorithm, formulated in nPCF^2, is shown in Figure 4.5c-e. It finds the normal form of a $\mathsf{nPCF}^{\mathsf{tdpe}}$-term $E : \sigma^{\mathfrak{d}}$ by applying a type-indexed extraction function $\Downarrow^\sigma$ ("reification") on a particular instantiation, called the *residualizing instantiation* $\{\!|E|\!\}_{\mathsf{ri}}$, of term E in the language nPCF^2. Being an instantiation, which maps static constructs to themselves, $\{\!|\cdot|\!\}_{\mathsf{ri}}$ makes the TDPE algorithm natively implementable in nPCF^Λ through the embedding $\{\!|-|\!\}_{\mathsf{n}\epsilon}$ of Section 4.4. Indeed, the composition of $\{\!|-|\!\}_{\mathsf{ri}}$ and the embedding $\{\!|-|\!\}_{\mathsf{n}\epsilon}$ is essentially the same as Filinski's direct formulation in the one-level language.

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

Lemma 4.11. *For all types* σ, $\mathsf{nPCF} \vdash \rhd |\Downarrow^\sigma| = \lambda x.x : \sigma \to \sigma$ *and* $\mathsf{nPCF} \vdash \rhd |\Uparrow_\sigma| = \lambda x.x : \sigma \to \sigma$.

The lemma captures the intuition of TDPE as two-level η-expansion, as Danvy stated in his initial presentation of TDPE [12].

Theorem 4.12 (Semantic correctness of TDPE). *If* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$ *and* $\mathsf{nPCF}^2 \vdash \mathsf{NF}(E) \Downarrow \mathcal{O}$, *then* $\mathsf{nPCF} \vdash \rhd |\mathcal{O}| = |E| : \sigma$.

Proof. A simple induction on E establishes that $\mathsf{nPCF} \vdash \rhd |\{\!|E|\!\}_{\mathsf{ri}}| = |E| : \sigma$,

which has the immediate corollary that $\mathsf{nPCF} \vdash \rhd |\mathsf{NF}(E)| = |E| : \sigma$. We then apply Theorem 4.3. □

4.6 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 4.1), we can already infer some intensional properties about the output of TDPE: It does not contain static constructs, and it is typable in nPCF. Furthermore, a quick inspection of the TDPE algorithm reveals that it will never construct a β-redex in the output—since there is no way to pass a dynamic λ-abstraction to the $\uparrow$ function. Indeed, an ad-hoc native implementation can be easily refined to express this constraint by changing the term type Λ. To capture that the output is fully η-expanded by typing, however, appears to require dependent types for the term representation.[7]

To show that the output of TDPE is always in long $\beta\eta$-normal form, i.e., typable according to the rules in Figure 4.6 (directly taken from Filinski [31]), we can take inspiration from the evaluation of nPCF^2-terms of type $\bigcirc\sigma$. Type preservation shows that evaluating these terms always yields a value of type $\bigcirc\sigma$, which corresponds to a well-typed nPCF-term. Similarly, to show that evaluating

[7] On the other hand, through some extra reasoning on the way the two-level program is written, it is possible to prove that the output is fully η-expanded in such a setting, as done by Danvy and Rhiger recently [23].

$\mathsf{NF}(E)$ always yields long $\beta\eta$-terms, we can refine the dynamic typing rules of nPCF^2, so that values of code type correspond to terms in long $\beta\eta$-normal form, and then we show that (1) evaluation preserves typing in the new type system; and (2) the term $\mathsf{NF}(E)$ 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 4.7. Briefly, we attach the sort of the judgment, atomic *at* or normal form *nf*, with the code type, and add another code type $\bigcirc^{var}(-)$ for variables in the context. This way, evaluation of static β-redexes will not substitute the wrong sort of syntactic phrase and introduce ill-formed code. The type system is a refinement of the original type system in the sense that all the new dynamic typing rules are derivable in the original system, if we ignore the new "refinement" tags (*at*, *nf*, *var*), and hence any term typable in the new type system is trivially typable in the original one.

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

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

Figure 4.6: Inference rules for terms in long $\beta\eta$-normal form

Theorem 4.13 (Refined type preservation). *If* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$, *then* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \tau$.

Theorem 4.14 (Normal-form code types). *If* V *is an* nPCF^2*-value (Figure 4.2), then*

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

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

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

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

(Dynamic)

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

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

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

Figure 4.7: nPCF^{2}-terms that generate code in long $\beta\eta$-normal form

(2) if $\mathsf{nPCF}^{2} \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \bigcirc^{nf}(\sigma)$*, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^{nf} |\mathcal{O}| : \sigma$*.*

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

Lemma 4.15. *(1) The extraction functions (Figure 4.5c) have the following normal-form types (writing* $\sigma^{\bigcirc \mathsf{nf}}$ *for* $\sigma\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*).*

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

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

Theorem 4.16. *If* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$*, then* $\mathsf{nPCF}^{2} \vdash \blacktriangleright \mathsf{NF}(E) : \bigcirc^{nf}(\sigma)$*.*

Corollary 4.17 (Syntactic correctness of TDPE). *For* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \triangleright E : \sigma^{\mathfrak{d}}$, *if* $\mathsf{nPCF}^{2} \vdash \mathsf{NF}(E) \Downarrow V$, *then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\mathsf{nPCF} \vdash \Delta \triangleright^{nf} |\mathcal{O}| : \sigma$.

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 [33, 90]. 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.

4.7 The general framework

In this chapter, 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) 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 [33]). By establishing the **type preservation** theorem for the type system, we further have that code-typed programs generate only well-typed terms.

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

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

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

In Chapter 4, the call-by-name, effect-free setting of $\mathsf{nPCF}^{\mathsf{2}}$ 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.

Chapter 5

The call-by-value two-level language vPCF²

In this section we design a two-level language vPCF² with Moggi's computational λ-calculus λ_c [74] as its object language in such a way that the language has the desired properties that we identified in Section 4.7 (Section 5.1). These properties are used to give a clean account of call-by-value TDPE (Section 5.3).

5.1 Design considerations

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

The problematic rule is that of static function applications:

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

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

This problem can be solved by using the canonical technique of let-insertion in partial evaluation [9]: 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 $ⓥ\sigma$ 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 $ⓔ\sigma$ to indicate possible computational effects, can be coerced into type $ⓥ\sigma$ with a "trivialization" operator #, which performs let-insertion.

Types $\tau ::= \theta \mid ⓔ\sigma$

$\theta ::= \mathsf{b} \mid ⓥ\sigma \mid \theta \to \tau$ (substitution-safe types)

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

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

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

$\mid E_1 \otimes E_2 \mid \$_{\mathsf{b}} E \mid \underline{\lambda} x.E \mid E_1 \underline{@} E_2 \mid \underline{d}$

$\mid \underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2 \mid \#E$

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

(Static)

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

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

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

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

(Dynamic)

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

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

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

Figure 5.1: The type system of vPCF^2

5.2 Syntax, semantics, and properties

The syntax and evaluation semantics of vPCF^2 are shown in Figure 5.1 and 5.2. Again, the languages are parameterized over a signature of typed constants. Due to the differences between call-by-name and call-by-value languages, the type of many important constants might differ: For example, for the conditional construct, we should have object-level constants $\mathsf{if}_\sigma : \mathsf{bool} \to (\mathsf{unit} \to \sigma) \to (\mathsf{unit} \to \sigma)$ (where unit is the standard unit type, which we omit from our language specification for the sake of brevity).

Note that the type θ of a function argument must be "substitution-safe", i.e., it cannot take the form $\textcircled{e}\sigma$. The corresponding one-level language vPCF is an instance of the λ_c-calculus: Its syntax is the same as nPCF of Figure 4.3, 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 vPCF^2, 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 beginning, and insert the accumulated bindings at the end), and for the trivialization operator # (which inserts a binding to the accumulator).

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

Let us examine the desired properties.

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 [44].

Definition 5.1 (Binder-term-in-context). *For a binder* $B \equiv (x_1 : \sigma_1 = \mathcal{O}_1, \ \ldots, x_n{:}\sigma_n = \mathcal{O}_n)$, *we write* $\Gamma \rhd [B]E : \tau$ *if* $\Gamma, x_1 : \textcircled{v}\sigma_1, \ldots, x_{i-1} : \textcircled{v}\sigma_{i-1} \rhd \mathcal{O}_i : \textcircled{e}\sigma_i$ *for all* $1 \leq i \leq n$, *and* $\Gamma, x_1 : \textcircled{v}\sigma_1, \ldots, x_n : \textcircled{v}\sigma_n \rhd E : \tau$.

Theorem 5.2 (Type preservation). *If* $\textcircled{v}\Delta \rhd [B]E : \tau$ *and* $\Delta \rhd [B]E \Downarrow [B']V$, *then* $\textcircled{v}\Delta \rhd [B']V : \tau$.

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

Definition 5.3 (Observation of complete program). *For a complete program* $\rhd E : \textcircled{e}\sigma$, *we write* $E \searrow \underline{\textbf{let}}^* \ B \ \underline{\textbf{in}} \ \mathcal{O}$ *if* $\rhd [\cdot]E \Downarrow [B]\mathcal{O}$.

Corollary 5.4 (Type preservation for complete programs). *If* $\rhd E : \textcircled{e}\sigma$ *and* $E \searrow \mathcal{O}$, *then* $\rhd \mathcal{O} : \textcircled{e}\sigma$.

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

Theorem 5.6 (Annotation erasure for complete programs). *If* $\mathsf{vPCF}^{2} \vdash \rhd E : \textcircled{e}\sigma$ *and* $\mathsf{vPCF}^{2} \vdash E \searrow \mathcal{O}$, *then* $\mathsf{vPCF} \vdash\ \rhd |E| = |\mathcal{O}| : \sigma$.

Native embedding: Without going into detail, we remark that vPCF^{2} has a simple native embedding $\{\!|-|\!\}_{\mathsf{v}\epsilon}$ into $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$, a CBV language with a term type and a state that consists of two references cells: We use one to hold the bindings and the other to hold a counter for generating fresh variables. As such, the language $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$ is a subset of ML; the language vPCF^{2} can thus be embedded into ML, with dynamic constructs defined as functions.[1] The correctness proof for the embedding is by directly relating the derivation of the evaluation from a term E, in vPCF^{2}, and the derivation of the evaluation from its translation $\{\!|E|\!\}_{\mathsf{v}\epsilon}$, in $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$. The details of the native embedding and the accompanying correctness proof, again, are available in Appendix C.

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

The problem specification of CBV TDPE is similar to the CBN TDPE, where the semantics is given by a translation into vPCF instead of nPCF. We only need to slightly modify the original formulation by inserting the trivialization

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

operators # at appropriate places, so that the two-level program $\mathsf{NF}(E)$ type checks in vPCF^2. The call-by-value TDPE algorithm thus formulated is shown in Figure 5.3. 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 4.12. Composing with the native embedding $\{\!|-|\!\}_{\mathsf{v}\epsilon}$, we have an efficient implementation of this formulation—which is essentially the call-by-value TDPE algorithm that uses state-based let-insertion [97]; see also Filinski's formal treatment [32].

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 5.4 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 5.5, which specify the λ_c-normal forms [32]. 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 $\textcircled{e}^{bd}(\sigma)$.

Definition 5.7 (Refined binder-term-in-context). *For a binder* $B \equiv (x_1 : \sigma_1 = \mathcal{O}_1, \cdots, x_n : \sigma_n = \mathcal{O}_n)$, *we write* $\Gamma \blacktriangleright [B]E : \tau$ *if* $\Gamma, x_1 : \textcircled{v}^{var}(\sigma_1), \ldots, x_{i-1} :$

$\textcircled{v}^{var}(\sigma_{i-1}) \blacktriangleright \mathcal{O}_i : \textcircled{e}^{bd}(\sigma_i)$ *for all* $1 \leq i \leq n$, *and* $\Gamma, x_1 : \textcircled{v}^{var}(\sigma_1), \ldots, x_n : \textcircled{v}^{var}(\sigma_n) \blacktriangleright E : \tau$.

Theorem 5.8 (Refined type preservation). *If* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B]E : \tau$ *and* $\mathsf{vPCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$, *then* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B']V : \tau$.

Corollary 5.9 (Refined type preservation for complete programs). *If* $\blacktriangleright E : \textcircled{e}^{nc}(\sigma)$ *and* $E \searrow \mathcal{O}$, *then* $\blacktriangleright \mathcal{O} : \textcircled{e}^{nc}(\sigma)$.

Theorem 5.10 (Normal-form code types). *If* V *is an* vPCF^2*-value (Figure 5.1), and* $\mathsf{vPCF}^2 \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright V : \textcircled{v}^{X}(\sigma)$ *where* X *is av, nv, bd, or nc, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^X |\mathcal{O}| : \sigma$.

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 5.11. *(1) The extraction functions (Figure 5.3c) have the following normal-form types (writing* $\sigma^{\bigcirc \mathsf{nv}}$ *for* $\sigma\{\textcircled{v}^{nv}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*.)*

$$\blacktriangleright \Downarrow^{\sigma} : \sigma^{\bigcirc \mathsf{nv}} \to \textcircled{v}^{nv}(\sigma), \blacktriangleright \Uparrow_{\sigma} : \textcircled{v}^{av}(\sigma) \to \sigma^{\bigcirc \mathsf{nv}}.$$

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

Theorem 5.12. *If* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\eth}$, *then* $\mathsf{vPCF}^2 \vdash \blacktriangleright \mathsf{NF}(E) : \textcircled{v}^{nv}(\sigma)$.

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

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

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

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

We use the following abbreviations.

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

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

(Static)

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

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

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

(Dynamic)

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

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

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

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

Figure 5.2: The evaluation semantics of vPCF^2

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

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

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

b. Standard instantiation (TDPE-erasure)

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

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

c. Extraction functions $\Downarrow^\sigma$ and $\Uparrow_\sigma$:

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

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

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

d. Residualizing instantiation

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

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

e. The static normalization function

$$\mathsf{NF}(\rhd E : \sigma^\eth) = \Downarrow^\sigma \{\!|E|\!\}_{\mathsf{ri}} : \textcircled{e}\sigma$$

Figure 5.3: Call-by-value type-directed partial evaluation

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

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

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

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

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

(Dynamic)

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

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

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

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

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

Figure 5.4: vPCF^2-terms that generate code in λ_c-normal form

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

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

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

Figure 5.5: Inference rules for terms in λ_c-normal form

Chapter 6

Related work

The introduction (Chapter 3) of Part I 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.

6.1 Two-level formalisms for compiler construction

While Jones et al. [57, 58, 59] studied two-level languages mainly as meta-languages for expressing partial evaluators and proving them correct, Nielson and Nielson's work explored various other aspects and applications of two-level languages, such as the following ones.

- A formalism for components of compiler backend, in particular code generation and abstract interpretation, and associated analysis algorithms [79]. 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 [104]. 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 [80], provides a setting for comparing and relating such languages, and, on the prescriptive side [81], 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 Part I, 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 [65], viewed as an interpreter, into a compiler [77]. Mosses developed action semantics [78] 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 [68].

6.2 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 [37], but he treated fresh name generation informally. Moggi gave a formal proof, using a functor category semantics to model name generation [75]. 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 [31, 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 [105, 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 [83] 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.

6.3 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 [51]. 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 [5] 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 [62, 63], 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 [60].

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 Part I, 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.

6.3.1 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,[1] 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 [106].

Davies and Pfenning investigated multi-level languages through the Curry-Howard correspondence with modal logics: $\lambda^{\Box}$, which corresponds to intuitionistic modal logic S4, has the run construct, but it can only manipulates closed code fragment [27]; $\lambda^{\bigcirc}$, which corresponds to linear temporal logic, can manipulate open code fragment, but does not have the run construct [26]. 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 [76] combines $\lambda^{\Box}$ and $\lambda^{\bigcirc}$, by ensuring that the argument to the run-construct be properly closed. Calcagno et al. further studied how side effects can be added to Idealized MetaML while retaining type soundness [10].

While the development of various multi-level languages has been centered

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

on the conflicts of expressiveness and type soundness, other important aspects of multi-level languages, such as efficient code generation and formal reasoning, have not been much explored. Wickline et al. formalized an efficient implementation of $\lambda^{\Box}$ in terms of an abstract machine [106]. Taha axiomatized a fragment of Idealized MetaML, which can be used for equational reasoning [99].

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).

6.3.2 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 [35]) can be achieved with a two-level program—the staged interpreter. The erasure property reduces the correctness of automatic compilation to that of the interpreter.

Some applications are explained and analyzed using the technique of partial evaluation, but not realized through a dedicated partial evaluator. The one-pass CPS transformer of Danvy and Filinski is one such example. In this case, it is not a whole program, but the output of a transformation (Plotkin's CPS

translformation), to be binding-time analyzed. The explicit staging of two-level languages makes them the ideal candidate for describing such algorithms. The technique of Section 4.2, 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 [20]. We have also applied this technique to stage other monadic transformations (such as a state-passing-style transformation) into one-pass versions that avoid the generation of administrative redexes.

The two-level language vPCF^2 (and similarly, nPCF^2) can also be used to account for the self application of partial evaluators. Under the embedding translation, a vPCF^2-program becomes a one-level program in $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$, 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, Part I develops in detail the formulation and techniques for self-applying TDPE so as to produce efficient generating extensions; this recasts a joint work with Bernd Grobauer[38], using the framework developed here in Part I.

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 4.6 and 5.3) to guarantee that code generated by a certain (possibly third-party) program is typable in such a type system; this could lead to the addition of a code-generating dimension to the area of trusted computing.

Chapter 7

Concluding remarks for Part I

7.1 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 [26, 76, 79].

- 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, Part I 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 [12].

- 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 [31, 32], which have been proved from scratch using denotational methods.

7.2 Direction for future work

It would be interesting to see whether and how far our general framework (Section 4.7) can apply to other scenarios, e.g., where the object language is a concurrent calculus, equationally specified. As we have seen in Chapter 6, 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 Part I, 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.

Part II

Encoding types

Chapter 8

Introduction to Part II

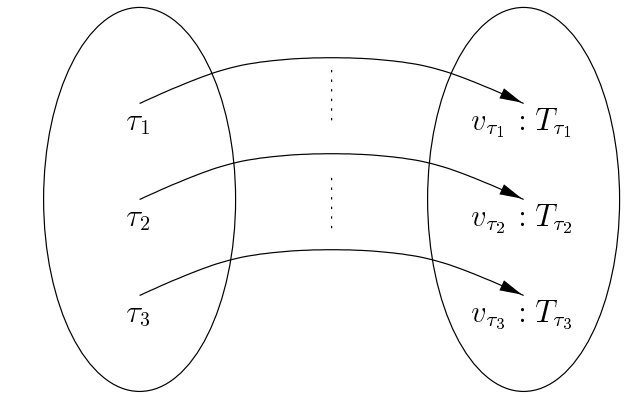
Over the last two decades, the Hindley-Milner type system [47, 70] evolved into the most popular type basis of functional languages. For example, it underlies several major higher-order, statically typed functional programming languages, such as ML [71] and Haskell [85]. Among other reasons, this popularity can be attributed to (1) static typing, which serves as a static debugging facility, and (2) implicit polymorphism (allowed by the *principal typing* scheme), which removes the burden of pervasive explicit type annotations. The simplicity of the type system, however, also restricts the class of typeable programs. For example, it is impossible to examine the type of a value at run-time, as in a dynamically typed language such as Scheme [60].

Functions that take type arguments and return values of possibly different types accordingly appear frequently in the abstract formulations of certain algorithms. These functions form an interesting class of programs, which seem to be beyond the capability of the Hindley-Milner type system. In Part II, we for-

mulate such a function as a *type-indexed family of values*, i.e., a family of values indexed by one or more type argument(s). Figure 8.1 illustrates a type-indexed family v indexed by one type argument τ: $v = \{v_\tau\}_{\tau \in F}$, where τ ranges over a family F of types. For a given type τ, the corresponding value is v_τ of type T_τ.

Usually, the family F of types is inductively specified using a set of type constructors. Correspondingly, the F-indexed family v of values is defined by case analysis on the type constructions. Since all types are implicit in a language with Hindley-Milner type system, only value encodings of types, instead of types themselves, can serve as the arguments of a function that represents a type-indexed family. We can try to reduce case analysis on type constructions to case analysis on value constructions, by encoding the type arguments using inductive data types. This, however, does not solve the problem, because different branches of the case-expression might have different types, and hence the case-expression may not be typeable. A common strategy in such a situation is to use tagged inputs and outputs, which are of some inductive data types as well. However, this solution requires users to tag input values themselves, which is not only inconvenient, and even unusable for cases when verbatim values are required, but also "type-unsafe", in that a run-time exception might be raised due to unmatched tags.

The problem of programming with type-indexed families of values has exposed the limitations of the Hindley-Milner type system, and it has motivated a line of research that explores more expressive type systems, notably intensional type analysis [42] and polytypic typing [50]. This article, in contrast, investigates what can be done *within* the framework of the Hindley-Milner type system. We



A type-indexed family v of values is a function that maps a family of types τ to a family of values v_τ of types T_τ.

Figure 8.1: A type-indexed family of values

demonstrate our methods with ML, though the techniques are equally applicable to any other functional language based on the Hindley-Milner type system.

We first show that interpreting types τ using corresponding values v_τ gives a type-safe solution to the problem. Based on our approach to type encodings, examples ranging from a printf-like formatting function[1] to type-directed partial evaluation can be programmed in ML successfully. Their type safety is automatically ensured by the ML type system, statically.

The aforementioned type encoding is application-specific, or *value-dependent*, i.e., the type encoding is tied to a specific family of values. Such a type encoding is not suitable in modular programming practice: one should be able to program different type-indexed families that share the same family of type indices

[1] Initially devised by Danvy [13].

separately, and combine them later. It is therefore interesting to find a method of type encoding that is independent of any particular type-indexed family. A value-independent encoding of a specific type τ can then be combined with the representation of a type-indexed family, say v, to deliver the value v_τ. We present two methods of creating such a value-independent type encoding:

1. A type-indexed family of values is specified as a tuple of value constructing functions, one for each possible type constructor, and the encoding of a specific type recursively selects and applies components from the tuple. This gives rise to a Martin-Löf-style encoding of inductive types. The encoding uses first-class polymorphism and higher-order polymorphism, and can be implemented using the higher-order module language of Standard ML of New Jersey [4].

2. A type τ is encoded as the embedding and projection functions between verbatim values, of type τ, and tagged values, of a universal data type U. To encode a specific value v_τ of a type-indexed family v, we can first define its equivalent value, whose type substitutes the universal data type for type τ, and then coerce it to the specific value of the indexed type. We show that this type encoding is universal, i.e., the coercion function can always be constructed from the embedding and projection functions of the indexed types.

In Chapter 9, we formalize the notion of type-indexed values, give examples, and discuss why it is difficult to program with them. In Chapter 10, we view type encodings as type interpretations, characterize requirements for

correct implementations of type-indexed values, and give a value-dependent approach to programming type-indexed values in ML. In Chapter 11, we present two approaches to value-independent type encodings, and argue that the second approach is universal and more practical. We discuss related work in Chapter 12 and conclude in Chapter 13.

Chapter 9

Type-indexed families of values

9.1 The notion

Type-indexed families of values (or just type-indexed families) are used in the formulation of algorithms given in a type-indexed (or type-directed) fashion. Depending on input type arguments, specific values from the same family could have different types. For brevity, we mainly consider such families indexed by only one type argument. Multiple type arguments can be dealt with by bundling all type indices into one type index. This solution, however, could lead to code explosion. We will come back to a practical treatment for dealing with multiple type arguments in Section 11.4.

A type-indexed family is usually defined by induction on the type τ, i.e., it is specified in the form

$$v_\tau = e$$

where expression e performs a case analysis on type τ, and uses the values in-

dexed at the component types of type τ. This notion is captured in the following definitions.

Definition 9.1 (Family of types). *An (inductive) family of types is inductively constructed in the following form:*

$$\begin{array}{rcl} \tau & = & c_1(\tau_{11}, ..., \tau_{1m_1}) \\ & | & \ldots \\ & | & c_n(\tau_{n1}, ..., \tau_{nm_n}) \end{array} \tag{9.1}$$

where each c_i is a type constructor, representing a type construction in the underlying language (ML in our case), which builds a type τ using component types *τ_{i1} through τ_{im_i}. We write $F^{c_1,\cdots,c_n}$ for the family of types.*

In most of our examples, the constructors c_i are named as the type constructions they represent, such as $\times$ (product) and $\rightarrow$ (function). It is not uncommon, however, that the constructors c_i are named according to the application: In the string-formatting example (Section Section 10.4), intuitive formatting directives are used as the constructor names. For these cases, one can take the equivalent view that the constructors define an arbitrary syntactic inductive family, and there is a compositional mapping from this inductive family to the type it represents; we make this mapping explicit notationally whenever necessary.

Without loss of generality, we assume that the case analysis used in the definition of the type-indexed family occurs at the outer-most level of the right-hand-side expression. This assumption accommodates the following definition.

Definition 9.2 (Type-indexed family of values). *A family of values indexed by the family $F^{c_1,\cdots,c_n}$ of types is specified inductively in the following pattern-*

matching form:

$$\begin{aligned} v_{c_1(\tau_{11},\ldots,\tau_{1m_1})} &= e_1(v_{\tau_{11}},\ldots,v_{\tau_{1m_1}}) \\ &\vdots \\ v_{c_n(\tau_{n1},\ldots,\tau_{nm_n})} &= e_n(v_{\tau_{n1}},\ldots,v_{\tau_{nm_n}}) \end{aligned} \tag{9.2}$$

In the other words, specific values in the family are constructed inductively using the expressions e_1 to e_n. For a given type τ, we write v_τ to represent the (unique) term obtained by unfolding the specification according to τ. We require that each expression e_i be expressible in the underlying language, which is ML in our case; this condition guarantees that each v_τ must be an ML-expression. The types of each e_i, consequently, might contain type variables that are bound at the top level, i.e., the type must be rank-1 polymorphic type of the form $\forall\vec{\alpha}.\tau$.

It is the possible variation in the types of the values that makes programming with a type-indexed family challenging and interesting. We therefore should make precise the types of the values in a type-indexed family of values. We observe that, in the applications that we encounter, the types of the values in the same family stay essentially the *same*, modulo the occurrences of the type index. That is, there is a type Q with a free type variable α, such that the type of value v_τ is $T_\tau = Q\{\tau/\alpha\}$.[1] Returning to Definition 9.2, we can infer that the type scheme of the expression e_i should be instantiatable to $T_{\tau_1} \times \cdots \times T_{\tau_{m_i}} \to T_\tau = Q\{\tau_1/\alpha\} \times \cdots \times Q\{\tau_{m_i}/\alpha\} \to Q\{c_i(\tau_1,\ldots,\tau_{m_i})/\alpha\}$. Abstracting over the types τ_1 through τ_{m_i}, we can write the type scheme for a type-indexed family of values in the following form.

[1] The type Q could contain other free type variables, which, however, can only be used monomorphically.

Definition 9.3 (Type scheme). *A type-indexed family of values v, as given in Definition 9.2 (page 86), is assigned a* type scheme *as $v : \forall\alpha \in F^{c_1,\cdots,c_n}.Q$, if expression e_i has (Hindley-Milner) type scheme $\forall\alpha_1,\ldots,\alpha_{m_i}.(Q\{\alpha_1/\alpha\} \times \cdots \times Q\{\alpha_{m_i}/\alpha\}) \to Q\{c_i(\alpha_1,\cdots,\alpha_{m_i})/\alpha\}$ for all $1 \leq i \leq n$.*

9.2 Running examples

This section introduces some examples to demonstrate the challenges posed by type-indexed families, and later to illustrate our methods for programming with them.

9.2.1 List flattening and polytypic printing

The flatten program, which flattens arbitrary nested lists with integer elements, is a toy example often used to illustrate the intricacy of typing "typecases" (case studies on types) in languages with Hindley-Milner type systems. It can be written in a dynamically typed language like Scheme (where type testing is allowed) as:

$$\begin{array}{lcll} \mathsf{flatten}\ x & = & [x] & (\text{where } x \text{ is atomic}) \\ \mathsf{flatten}\ [x_1,\ldots,x_n] & = & (\mathsf{flatten}\ x_1) \oplus \cdots \oplus (\mathsf{flatten}\ x_n) & \end{array}$$

where $\oplus$ is the list concatenation operator. To write this function in ML, a natural solution is to use the ML data type mechanism to define a "list" data type, and use pattern matching facilities for case analysis. However, this requires a user to tag all the values, making it somewhat inconvenient to use. Is it possible to use verbatim values directly as the arguments? The term "verbatim

values" here refers to values whose types are formed using only native ML type constructors, and are hence free of user-defined value constructors.

Due to restrictions of the ML type system, a verbatim value of nested list type must be homogeneous, i.e., all members of the list must have the same type. In the particular case that members are lists themselves, they must have the same nesting depth. The flatten function for verbatim values can therefore be specified as a family of flatten functions indexed by the possible type argument τ.

Example 9.4 (flatten). *The family* $F^{\mathsf{int},\mathsf{list}}$ *of types is generated by the following grammar.*[2]

$$\tau \quad = \quad \mathsf{int} \mid \tau_1\ \mathsf{list}$$

The flatten function can then be specified as an $F^{\mathsf{int},\mathsf{list}}$*-indexed family of function values.*

$$\begin{array}{lcl} \mathsf{flatten} & : & \forall\alpha \in F^{\mathsf{int},\mathsf{list}}.\alpha \to \mathsf{int\ list} \\ \mathsf{flatten}_{\mathsf{int}}\ x & = & [x] \\ \mathsf{flatten}_{\tau_1\ \mathsf{list}}[x_1,\ldots,x_n] & = & (\mathsf{flatten}_{\tau_1}\ x_1) \oplus \cdots \oplus (\mathsf{flatten}_{\tau_1}\ x_n) \end{array}$$

This definition conforms to Definition 9.2 (page 86), since we can also write:

$$\begin{array}{lcl} \mathsf{flatten}_{\mathsf{int}} & = & e_1 \\ \mathsf{flatten}_{\tau_1\ \mathsf{list}} & = & e_2(\mathsf{flatten}_{\tau_1}) \end{array}$$

where

$$\begin{array}{ll} e_1 : \mathsf{int} \to \mathsf{int\ list} & = \lambda x.[x] \\ e_2 : \forall\alpha.(\alpha \to \mathsf{int\ list}) \to (\alpha\ \mathsf{list} \to \mathsf{int\ list}) & = \lambda f.\lambda[x_1,\ldots,x_n].f\ x_1 \oplus \cdots \oplus f\ x_n \end{array}$$

[2]It is for the ease of presentation that we use int for the base case here, instead of a universally quantified type variable. We discuss type variable and polymorphism in Section 10.5.

It should be easy to check that flatten *has the declared type scheme (Definition 9.3, page 88).*

Before trying to write the function flatten, let us analyze how it might be used. A first attempt is to make the input value (of some arbitrary homogeneously nested list type) the only argument. This would require that both the expression `flatten 5` and the expression `flatten [6]` type-check, so the function argument should be of a polymorphic type that generalizes both type `int` and type `int list`—this polymorphic type can only be a type variable α. But ML's parametric polymorphism disallows inspecting the type structure of a polymorphic value. Consequently, it is impossible to write function flatten with the value to be flattened as the only argument.

The next attempt is to use an extra argument for describing the input type, i.e., a value that encodes the type. We expect to rewrite the aforementioned function invocations as `flatten Int 5` and `flatten (List Int) [6]`, respectively. One might try to encode the type using a datatype as:

```
datatype typeExp = Int | List of typeExp
```

The fixed type `typeExp` of the type encoding, however, constrains the result of applying function flatten to the type encoding to a fixed ML type again. A simple reasoning like the one for the first attempt shows that it is still impossible to give a typeable solution in ML.

A similar, but probably more useful example of type-indexed family, is polytypic printing. The functionality of polytypic printing is to convert data of arbitrary verbatim types to a string representation suitable for output, accord-

ing to the type. For the type constructors, we consider integer, string, product, and list.

Example 9.5 (polytypic printing). *The family* $F^{\mathsf{int},\mathsf{str},\times,\mathsf{list}}$ *of types is generated by the following grammar.*

$$\tau \quad = \quad \mathsf{int} \mid \mathsf{str} \mid \tau_1 \times \tau_2 \mid \tau_1\ \mathsf{list}$$

Polytypic printing is specified as a $F^{\mathsf{int},\mathsf{str},\times,\mathsf{list}}$*-indexed family of functions.*

$$\begin{array}{lcl}
\mathsf{toStr} & : & \forall \tau \in F^{\mathsf{int},\mathsf{str},\times,\mathsf{list}}.\tau \to \mathsf{str} \\
\mathsf{toStr}_{\mathsf{int}}\, x & = & \mathsf{intToStr}\, x \\
\mathsf{toStr}_{\mathsf{str}}\, s & = & s \\
\mathsf{toStr}_{\tau_1 \times \tau_2}\, (x_1, x_2) & = & \text{“(”} \mathbin{\hat{}} (\mathsf{toStr}_{\tau_1}\, x_1) \mathbin{\hat{}} \text{“,”} \mathbin{\hat{}} (\mathsf{toStr}_{\tau_2}\, x_2) \mathbin{\hat{}} \text{“)”} \\
\mathsf{toStr}_{\tau_1\ \mathsf{list}}[x_1, \ldots, x_n] & = & \text{“[”} \mathbin{\hat{}} (\mathsf{toStr}_{\tau_1}\, x_1) \mathbin{\hat{}} \cdots \mathbin{\hat{}} \text{“,”} \mathbin{\hat{}} (\mathsf{toStr}_{\tau_1}\, x_n) \mathbin{\hat{}} \text{“]”}
\end{array}$$

where $\mathbin{\hat{}}$ *is the string concatenation function.*

Having specified polytypic printing, we can use it to define a C printf-style formatting function, also as a type-indexed family. The formatting function, however, is slightly more involved, in both conceptual terms and practical terms. In order not to distract the reader with its details at the current stage, we postpone its development to Section 10.4, where the basic programming technique has already been introduced.

9.2.2 Type-directed partial evaluation

Let us apply the type encoding technique to type-directed evaluation, the running example of this dissertation. Here in Part II, in order to make the development independent of Part I and the presentation concrete, we return to use ML

with exp as an informal two-level language, where names need to be generated explicitly. To recall from the informal introduction to TDPE in Section 2.4.2, we use an inductive type exp, which provides representation for generated code.

```
datatype exp = VAR of string
             | LAM of string * exp
             | APP of exp * exp
```

We write $\underline{\lambda}x.E$ as shorthand for `LAM(`x`,` E`)`, $E_1 \underline{@} E_2$ as shorthand for `APP(`E_1`,`E_2`)`, and an occurrence of $\underline{\lambda}$-bound variable x as shorthand for `VAR(`x`)`.

Example 9.6 (Type-directed partial evaluation). *The family* $F^{\mathsf{exp},\mathsf{func}}$ *of types is generated by the following grammar.*

$$\tau \quad = \quad \mathsf{exp} \;\mid\; \tau_1 \to \tau_2$$

The extraction function for type-directed partial evaluation is defined as two families of type-indexed functions $\downarrow$ *(reify) and* $\uparrow$ *(reflect) (Figure 9.1), which recursively call each other for the contravariant function argument. At first glance, their definitions do not fit into the canonical form of a type-indexed family (Definition 9.2, page 86); however, pairing the two functions at each type index puts the definition into the standard form of a type-indexed family (Figure 9.2).*

In the following, we write "$\bullet$" as a shorthand for the type exp.

It might be helpful, for a rough intuition of the TDPE algorithm, to work out some simple examples, such as $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}((\lambda x.\lambda y.x)\,(\lambda x.\lambda y.x))$ and $\downarrow^{(\bullet\to\bullet)\to(\bullet\to\bullet)}(\lambda f.\lambda x.f(f(x)))$; see Section 2.4 of this dissertation. Detailed accounts of type-directed partial evaluation can be found in the literature [15, 31, 38].

$$
\begin{array}{lllll}
\text{(reify)} & \downarrow^{\mathsf{exp}} v & = & v \\
& \downarrow^{\tau_1 \to \tau_2} f & = & \underline{\lambda} x.\downarrow^{\tau_2} (f\,(\uparrow_{\tau_1} x)) \\
& & & \text{(where } x \text{ is a fresh variable)} \\
\text{(reflect)} & \uparrow_{\mathsf{exp}} e & = & e \\
& \uparrow_{\tau_1 \to \tau_2} e & = & \lambda v_1.\uparrow_{\tau_2} (e \underline{@} (\downarrow^{\tau_1} v_1))
\end{array}
$$

Figure 9.1: Type-directed partial evaluation

$$
\begin{array}{lcll}
(\downarrow, \uparrow) & : & \forall \tau \in F^{\mathsf{exp},\mathsf{func}}.(\tau \to \mathsf{exp}) \times (\mathsf{exp} \to \tau) \\
(\downarrow, \uparrow)_{\mathsf{exp}} & = & (\lambda v.v, \lambda e.e) \\
(\downarrow, \uparrow)_{\tau_1 \to \tau_2} & = & \textbf{let} & (\downarrow^{\tau_1}, \uparrow_{\tau_1}) = (\downarrow, \uparrow)_{\tau_1} \\
& & & (\downarrow^{\tau_2}, \uparrow_{\tau_2}) = (\downarrow, \uparrow)_{\tau_2} \\
& & \textbf{in} & (\lambda f.\underline{\lambda} x.\downarrow^{\tau_2} (f\,(\uparrow_{\tau_1} x)), \\
& & & \lambda e.\lambda v.\uparrow_{\tau_2} (e \underline{@} (\downarrow^{\tau_1} v)) \\
& & & \text{(where } x \text{ is a fresh variable)}
\end{array}
$$

Figure 9.2: TDPE in the general form of type-indexed family

In his article [12], Danvy presents the Scheme code for this algorithm. He represents the type index as a value—which is an S-expression and is similar to a value of inductive data type in ML—thereby reducing type analysis to case analysis. A direct transcription of that program into an ML program,

however, would require the input arguments to be tagged. Such a solution is not satisfactory for the following reasons:

- Using type-directed partial evaluation, one expects to normalize a program in the source language with minimum modification. It is cumbersome for the user to tag/untag all the program constructs. A verbatim program is much preferable in this case.

- Unlike the function flatten, the function $\uparrow$ (reflect) requires the type argument explicitly. The type index τ only appears as the codomain of the function $\uparrow$, whereas its domain is always of type exp. For the same input expression, varying the type argument results in different return values of different types.

  Because explicit type arguments must be present, static type checking of ML cannot guarantee the consistency of the type argument and the tags attached to the input values cannot be guaranteed by static type checking of ML: run-time "type error" can arise in the form of a pattern mismatch exception. This problem is also present in the Scheme program.

Chapter 10

Type-indexed families as type interpretations

Our first approach to programming type-indexed families v is based on interpreting the types τ as the values v_τ indexed by these types.

10.1 Implementing indexed families with type encodings: the idea

As we argued in the list-flattening example (Section 9.2.1), if verbatim arguments are required for an ML function that represents a type-indexed family, then (1) a type encoding must be explicitly provided as an argument to the function, and (2) this type encoding cannot have a fixed type. Now that the type encodings, say E_τ for all $\tau \in F$, themselves must have different types, a reasonable choice of these types should make them reflect the types τ being encoded.

There are infinitely many types, therefore we should encode type constructors, such that the encoding of a type is constructed inductively by using the encoding of the type constructors. To be more precise, for each type constructor c of arity m (i.e., it constructs a type τ from types $\tau_1, \ldots, \tau_m$), its encoding E_c as a term (in ML) is a function that transforms the type encodings $E_{\tau_1}, \ldots, E_{\tau_m}$ to the type encoding E_τ. In other words, the encodings of inductively constructed types should form a particular syntactic interpretation (also called an instantiation), in the underlying language. If we use $\langle\!|u|\!\rangle$ instead of E_u to denote the interpretation, we can write down the requirements for the encodings:

$$\begin{array}{lrcl} \text{If} & \tau & = & c(\tau_1, \ldots, \tau_m) \\ \text{then} & \langle\!|\tau|\!\rangle & \equiv & \langle\!|c|\!\rangle(\langle\!|\tau_1|\!\rangle, \ldots, \langle\!|\tau_m|\!\rangle). \end{array}$$

This can be understood as requiring the interpretations of type and type constructors to form a homomorphism, i.e.,

$$\langle\!|c(\tau_1, \ldots, \tau_m)|\!\rangle \equiv \langle\!|c|\!\rangle(\langle\!|\tau_1|\!\rangle, \ldots, \langle\!|\tau_m|\!\rangle) \tag{10.1}$$

This way, for the encoding of a whole family of types, it suffices to give the encoding of the type constructors, since the encoding of every type is uniquely determined.

Definition 10.1 (Encoding a family of types). *The encoding of a family of types $F^{c_1,\cdots,c_n}$ (as given in Definition 9.1 (page 86)) is specified as the encoding of the constructors c_1 through c_n as ML-terms $\langle\!|c_1|\!\rangle$ through $\langle\!|c_n|\!\rangle$, such that the encoding $\langle\!|\tau|\!\rangle$ of every type, induced according to Equation (10.1), is typable.*

With such an encoding $\langle\!| \cdot |\!\rangle$ of a type family F, a F-index family v of values

can then be represented as a function f_v that takes the type encoding as an argument.

Definition 10.2 (Implementation of type-indexed families). *Let F be a family of types, v be an F-indexed family of values, and $\langle\!| \cdot |\!\rangle$ be an encoding of F (given on the constructors). A ML function f_v* implements *a type-indexed value v through the encoding $\langle\!| \cdot |\!\rangle$, if for all $\tau \in F$, the following equation holds.*

$$v_\tau = f_v \langle\!| \tau |\!\rangle \tag{10.2}$$

10.2 The ad-hoc approach

The task of finding the type encodings now boils down to finding suitable interpretations for the type constructors c_i. The close similarities between the general form of type-indexed values in the set of equations given by (9.2) on page 87 and the interpretation of type constructors in Equation (10.1) hints at an immediate solution to programming with type-indexed families: We can interpret a type τ as the corresponding value v_τ, which is achieved by interpreting the type construction c_i using the value construction e_i in the set of equations given by (9.2) on page 87.

Proposition 10.3 (Ad-hoc encoding). *Let $F^{c_1,\ldots,c_n}$ be a family of types and v an $F^{c_1,\ldots,c_n}$-indexed family of values, with associated data as presented in Definition 9.2 (page 86), i.e., type constructors c_i and corresponding data constructions e_i. The following equation defines a type encoding for F.*

$$\langle\!| c_i |\!\rangle \triangleq e_i$$

It induces the following encoding of types:

$$\langle\!|\tau|\!\rangle \equiv v_\tau$$

Proof. That $\langle\!|\tau|\!\rangle \equiv v_\tau$ follows immediately from Equation (9.2) on page 87 and Equation (10.1) on page 96. The typability of v_i implies the typability of $\langle\!|\tau|\!\rangle$. □

This type encoding is ad hoc, in the sense that it is specific to the F-indexed family v; the implementation of family v, then, is immediate.

Theorem 10.4. *Let $F^{c_1,\ldots,c_n}$ be a family of types and v a $F^{c_1,\ldots,c_n}$-indexed family of values, with associated data as presented in Definition 9.2 (page 86). The identity function $f_v \triangleq \lambda x.x$ implements v through the ad-hoc encoding $\langle\!|c_i|\!\rangle \triangleq e_i$.*

10.3 Examples

Let us demonstrate the ad-hoc type-encoding method with the running examples.

Example 10.5 (flatten). *The definition of the function* flatten *(Example 9.4, page 89) gives rise to the following interpretations of type constructions:*

$$\begin{aligned}
\langle\!|.|\!\rangle &: \forall\tau \in F^{\mathsf{int},\mathsf{list}}.\tau \to \mathsf{int\ list} \\
\langle\!|\mathsf{int}|\!\rangle &= \lambda x.[x] \\
\langle\!|\alpha\ \mathsf{list}|\!\rangle &= \lambda[x_1,\ldots,x_n].\langle\!|\alpha|\!\rangle x_1 @ \cdots @ \langle\!|\alpha|\!\rangle x_n
\end{aligned}$$

A direct coding of these interpretations of type construction as ML functions leads to the program in Figure 10.1.

Since we choose the ML function names to be the type constructors they interpret, a type argument, e.g., `List (List Int)`*, already has the value of*

$$\langle\!|(\mathsf{int\ list})\ \mathsf{list}|\!\rangle = \mathsf{flatten}_{(\mathsf{int\ list})\ \mathsf{list}},$$

```
val Int: int -> int list                                          (* ⟨|int|⟩ *)
    = fn x => [x]
fun List (T:'a -> int list): ('a list -> int list)                (* ⟨|list|⟩ *)
    = fn l => foldr (op @) [] (map T l)
fun flatten T = T                                  (* f_flatten ≜ λx.x *)
```

Figure 10.1: flatten in ML: ad-hoc encoding

and function `flatten` *can be defined just as the identity function. As desired, the function takes verbatim values as input. For example, the expression*

```
flatten (List (List Int)) [[1, 2], [], [3], [4, 5]]
```

evaluates to `[1,2,3,4,5]`*.*

This example exhibits the basic pattern of the ad-hoc approach to programming with a type-indexed family: for each type constructor, we bind to its corresponding ML name the expression e_i through a value or function definition. Their associated types, as given in Definition 9.3 (page 88), are rank-1 and therefore accepted by the ML type system.[1]

The same method works for the examples of polytypic printing and type-directed partial evaluation.

Example 10.6 (polytypic printing). *Figure 10.2 shows the ML implementation of polytypic printing, as formulated in Example 9.5 (page 91), using the type encoding* $\langle\!|\tau|\!\rangle \triangleq \mathsf{toStr}_\tau$.

As an example, evaluating the expression

```
toStr (List (Pair Str (List Str)))
```

[1] The type annotations are not necessary, since the ML type system infers the most general type scheme anyway; we include them for the sake of clarity.

```
type 'a ts = 'a -> string                                  (* Type scheme: α → str *)
val Int: int ts                                                        (* ⟨|int|⟩ *)
    = fn n => Int.toString n
fun Str: string ts                                                     (* ⟨|str|⟩ *)
    = fn s => s
fun Pair (toStr1: 'a ts) (toStr2: 'b ts)                               (* ⟨| × |⟩ *)
    : ('a * 'b) ts
    = fn (x1: 'a, x2: 'b) =>
         "(" ^ (toStr1 x1) ^ ", " ^ (toStr2 x2)  ^ ")"
fun List (toStr: 'a ts): ('a list ts)                                  (* ⟨|list|⟩ *)
    = fn (l: 'a list) =>
      let fun mkTail []  = "]"
            | mkTail [e] = (toStr e) ^ "]"
            | mkTail (e :: el)
              = (toStr e) ^ ", " ^ (mkTail el)
      in "[" ^ (mkTail l)
      end
fun toStr T = T                                        (* f_toStr ≜ λx.x *)
```

Figure 10.2: Polytypic printing in ML: ad hoc encoding

```
[("N", ["Prince", "8", "14"]),
 ("P", ["Newport", "Christopher", "9"])]
```

yields `"[(N, [Prince, 8, 14]), (P, [Newport, Christopher, 9])]"`.

Example 10.7 (type-directed partial evaluation). *Figure 10.3 shows the ML implementation of type-directed partial evaluation, as described in Example 9.6 (page 92) and formulated as a type-indexed family of values in Figure 9.2 (page 93), using the type encoding* $\langle\!|\tau|\!\rangle = (\downarrow, \uparrow)_\tau$. *The structure* `Gensym` *provides a function* `new` *for generating fresh names using a counter, and a function* `init` *to initialize this counter.*

As an example, the expression

```
reify_init (a' -> a' -> a' -> a')
```

```
datatype exp = VAR of string                                          (* exp *)
             | LAM of string * exp
             | APP of exp * exp

type 'a rr = ('a -> exp) * (exp -> 'a)
                                    (* Type scheme: (α → exp) × (exp → α) *)

infixr 5 -->
val a': exp rr                                                        (* ⟨exp⟩ *)
    = (fn v => v, fn e => e)
fun (T1 as (reif1, refl1): 'a rr) -->                                 (* ⟨ → ⟩ *)
    (T2 as (reif2, refl2): 'b rr): ('a -> 'b) rr
    =  (fn (f: 'a -> 'b) =>
           let val x = Gensym.new()
           in  LAM(x, reif2 (f (refl1 (VAR x))))
           end,
        fn (e: exp) =>
           fn (v: 'a) => refl2 (APP(e, reif1 v)))
fun reify (T as (reif_T, refl_T)) = reif_T                            (* f↓ *)
fun reify_init T v = (Gensym.init(); reify T v)
                                    (* reify, with name counter initialized *)
```

Figure 10.3: Type-directed partial evaluation in ML

```
((fn x => fn y => x) (fn x => fn y => x))
```

evaluates to `LAM("x1", LAM("x2", LAM("x3", VAR "x2")))`*, which represents the* λ*-expression* $\lambda x_1.\lambda x_2.\lambda x_3.x_2$.

10.4 Printf-style String formatting

We can apply the ad hoc type encoding method to program a type-safe formatting function in the style of the C printf function. In fact, this is an example where it is more natural not to view the indices as types directly, but to view them as syntactic phrases that are translated to the types that they represent under a compositional mapping, say $\mathcal{R}$.

We consider the formatting specification as a sequence of field specifiers. The grammar of formatting specification is given below:

$$\begin{aligned} \mathit{Spec} &::= \texttt{NIL} \mid \mathit{Field} :: \mathit{Spec} \\ \mathit{Field} &::= \texttt{LIT}\ s \mid \texttt{\%}\ \tau \end{aligned}$$

where s is a string literal and `%` τ specifies an input field argument of type τ. We want to write a function format such that, for instance, the expression

```
format (% Str ++ LIT " is " ++ % Int ++ LIT "-years old.")
       "Mickey" 80
```

evaluates to the string `"Mickey is 80-years old."`.

Function format is indexed by a formatting specification fs. A specialized format_{fs} has type $\tau_1 \to \tau_2 \ldots \to \tau_n \to \mathsf{str}$, where τ_i's are from all the field specifiers "`%` τ_i" in the specification fs in the order of their appearance. We make use of an auxiliary function format', which introduces one extra argument b as a string buffer; the function appends its output to the end of this input string buffer to build the output string. The functions format and format' can be formulated as follows.

Example 10.8. *The syntactic family F^{fs} of formatting specifications is given by the following grammar (in concrete syntax).*

$$fs ::= \texttt{NIL} \mid \texttt{LIT}\ s :: fs' \mid \texttt{\%}\ \tau :: fs' \quad (\tau \in F^{\mathsf{toStr}})$$

Here, the 0-ary constructor `NIL` *and the binary constructor* :: *are used for building sequences, while the unary constructors* `LIT` s *and* `%` τ *are used for building individual field specifiers.*

A formatting specification fs determines the type of format_{fs}*, through the compositional translation* $\mathcal{R}$*, defined as follows:*

$$\begin{aligned}
\mathcal{R}(\texttt{NIL}) &= \mathsf{str} \\
\mathcal{R}(\texttt{LIT}\ s :: fs') &= \mathcal{R}(fs') \\
\mathcal{R}(\texttt{\%}\ \tau :: fs') &= \tau \to \mathcal{R}(fs')
\end{aligned}$$

We specify format' *as an* F^{fs}*-indexed family via* $\mathcal{R}$*, and use it to define* format*.*

$$\begin{aligned}
\mathsf{format}' &: \forall fs \in F^{fs}.\mathsf{str} \to \mathcal{R}(fs) \\
\mathsf{format}'_{\texttt{NIL}} b &= b \\
\mathsf{format}'_{\texttt{LIT}\ s::fs'} b &= \mathsf{format}'_{fs'}(b \,\hat{}\, s) \\
\mathsf{format}'_{\texttt{\%}\ \tau::fs'} b &= \lambda(x:\tau).\mathsf{format}'_{fs'}(b \,\hat{}\, \mathsf{toStr}_\tau x)
\end{aligned}$$

$$\begin{aligned}
\mathsf{format} &: \forall fs \in F^{fs}.\mathcal{R}(fs) \\
\mathsf{format}_{fs} &= \mathsf{format}'_{fs}(\text{“ ”})
\end{aligned}$$

where the type-indexed family toStr *is from Example 9.5 (page 91).*

Following the ad-hoc method, we should interpreter each individual field specification f ($\texttt{LIT}\ s$ or $\texttt{\%}\ \tau$), which is a constructor for formatting specifications. The function $\langle\!|f|\!\rangle$ should be a transformer from format'_{fs} to $\mathsf{format}'_{f::fs}$, i.e.,

$$\mathsf{format}'_{f::fs} = \langle\!|f|\!\rangle\ \mathsf{format}'_{fs}$$

We obtain the interpretation of individual field specifiers by abstracting over format'_{fs}:

$$\begin{aligned}
\langle\!|\texttt{LIT}\ s|\!\rangle &= \lambda\mathsf{format}'_{fs}.\lambda b.\mathsf{format}'_{fs}(b \,\hat{}\, s) \\
\langle\!|\texttt{\%}\ \tau|\!\rangle &= \lambda\mathsf{format}'_{fs}.\lambda b.\lambda(x:\tau).\mathsf{format}'_{fs}(b \,\hat{}\, \mathsf{toStr}_\tau x)
\end{aligned}$$

On the practical side, it is undesirable to use the field specifications as nested prefix constructors that build formatting specifications from `NIL`. For this purpose, we define a function `++` to compose such transformers (similar to the function append for lists), and we can define the function format to take such specification transformer, instead of a specification, and to supply the interpretation of the empty field specification $\langle\!|\texttt{NIL}|\!\rangle = \mathsf{format}'_{\texttt{NIL}}$, along with an empty string as the initial buffer. Putting everything together, we have the ML implementation of the string formatter in Figure 10.4; it uses the implementation of toStr in Figure 10.2 (page 100).

```
infix 5 ++
type 'a fmt = string -> 'a
                                    (* Type scheme (of format'): str → R(fs) *)

fun LIT s: ('a fmt -> 'a fmt)                                    (* ⟨| LIT s|⟩ *)
    = fn (p: 'a fmt) => fn b => p (b ^ s)
fun % (toStr_t: 'c ts): ('a fmt -> ('c -> 'a) fmt)               (* ⟨| % τ|⟩ *)
    = fn (p: 'a fmt) => fn b =>
         fn (x: 'c) => p (b ^ toStr_t x)
fun f1 ++ f2 = f1 o f2
val NIL: (string fmt)                                            (* ⟨|NIL|⟩ *)
    = fn b => b
fun format (ftrans: string fmt -> 'a fmt)
    = ftrans NIL ""
```

Figure 10.4: printf-style formatting in ML: ad-hoc encoding

Unlike the C `printf` function, the above ML implementation is type-safe, in that compilation would reject mismatched field specification and input argument. For example, the type of the expression

```
format (% Int ++ LIT ":  " ++ % Str)
```

is $\mathsf{int} \rightarrow \mathsf{str} \rightarrow \mathsf{str}$, which ensures that exactly two arguments, one of type int, the

other of type str, can be supplied.

The printf example also showcases the power of a higher-order functional language with the possibility of building constructing field specifiers for compound types, through toStr. The following expression, following Example 10.6 (page 99), illustrates this flexibility.

```
format (LIT "MTA/NJT: " ++ %(List (Pair Str (List Str))))
        [("N", ["Prince", "8", "14"]),
         ("P", ["Newport", "Christopher", "9"])]
```

It should be clear that for any given type τ, we can have different functions to translate a value of type τ to its string representation. By defining more complicated field specifiers, which, e.g., allow variations in paddings, delimiters for the layout of compound types, and by refining the output type from string to more sophisticated formatting objects, we can construct sophisticated pretty-printers like John Hughes's [49], through the convenient printf-style interface.

10.5 Variations

The ad-hoc method easily adapts to several variations of type-indexed families.

Multiple translations of indices In the examples up until now, the type schemes either uses the indices directly, or a single translation of them (such as $\mathcal{R}$ in the string-formatting example). We shall see that having multiple translations of the indices occurring in the type scheme does not add to the complexity.

Other type constructions The type constructions used in the earlier examples include product, function, and list. One might wonder to what other type constructions the ad-hoc method is applicable. In fact, since we left unspecified the type constructions in our formulation of type-indexed families and their implementations, type-indexed families with any type constructions can be implemented using the ad-hoc method, as long as they can be cast into the form of Definition 9.2 (page 86) (where e_i's could be rank-1 polymorphic); we shall see an example that uses both the reference type and the continuation type (Example 10.9). Type constructions that bind type variables, however, do not conform to our formulation of type-indexed families.

- Polymorphic types: suppose that the universal quantifier $\forall\beta$ could be used as a constructor. It should construct from a type τ parameterized over a free type variable β (or, equivalently, a type constructor T such that $\tau = T(\beta)$), the type $\forall\beta.\tau$. A possible type for the corresponding value construction $e_{\forall\alpha.}$ could be $(\forall T : F \to F.(\forall\beta.Q\{T(\beta)/\alpha\} \to Q\{\forall\beta.T(\beta)/\alpha\}))$. Unfortunately, the polymorphism of this type is not rank-1, and it is higher-order; therefore universal qualification can not be used as a type construction.

  This analysis, however, does not rule out using a type variable as a base type (0-ary type constructor), universally quantified over at the top level of the type expression. In fact, in the flatten example, replacing the base type int by a type variable would cause no problem.

  Instead of performing analysis over the universal quantifier, one might want to parameterize an index over another index or a constructor. There is no

problem with abstracting over an index, since the corresponding encoding is used only monomorphically is building other type encodings. For an example, we can code $\lambda\tau.\tau \times \tau$ list as `fn t => Pair t (List Int)` for the polytypic printing example (Example 10.6 (page 99)). In contrast, to abstract over a constructor is generally not permissible, since we want to use the encoding of type constructors polymorphically in building type encodings. Again taking the polytypic printing example, coding $\lambda c.c((c(\mathsf{int}))\ \mathsf{list})$ as `fn x => x (List (x Int))` results in a type error.

- Recursive types: as we have seen in the flatten example, we can use a recursive type, such as list, in the type construction; we can as well replace list with a user-defined recursive type, such as $Tree$. On the other hand, by the same reason we cannot perform type analysis over a universal quantifier, we cannot perform type analysis over a recursive type.

Let us demonstrate some of these possible variations through another example: ironing (or, indirection elimination), which eliminates indirections of either reference type or double-continuation type. It showcases not only some extra type constructions, but also the use of two separate translations from the indices, one for the domain type, and one for the codomain type.

Example 10.9 (iron). *The family* F^{iron} *of indexes is generated by the following grammar.*

$$\tau \quad = \quad \mathsf{int} \mid \tau_1\ \mathsf{list} \mid \tau_1 \times \tau_2 \mid \tau_1\ \mathsf{ref} \mid \neg\neg(\tau_1)$$

The indices are almost the domain types, except for the constructor $\neg\neg$*, which is mapped into a double-continuation construction. The indices are mapped into*

the domain types and codomain types through the compositional translations $\mathcal{R}_I$ *and* $\mathcal{R}_O$*, respectively.*

$$
\begin{array}{lcl}
\mathcal{R}_I(\mathsf{int}) & = & \mathsf{int} \\
\mathcal{R}_I(\tau_1\ \mathsf{list}) & = & \mathcal{R}_I(\tau_1)\ \mathsf{list} \\
\mathcal{R}_I(\tau_1 \times \tau_2) & = & \mathcal{R}_I(\tau_1) \times \mathcal{R}_I(\tau_2) \\
\mathcal{R}_I(\tau_1\ \mathsf{ref}) & = & \mathcal{R}_I(\tau_1)\ \mathsf{ref} \\
\mathcal{R}_I(\neg\neg(\tau_1)) & = & \mathcal{R}_I(\tau_1)\ \mathsf{cont\ cont}
\end{array}
$$

$$
\begin{array}{lcl}
\mathcal{R}_O(\mathsf{int}) & = & \mathsf{int} \\
\mathcal{R}_O(\tau_1\ \mathsf{list}) & = & \mathcal{R}_O(\tau_1)\ \mathsf{list} \\
\mathcal{R}_O(\tau_1 \times \tau_2) & = & \mathcal{R}_O(\tau_1) \times \mathcal{R}_O(\tau_2) \\
\mathcal{R}_O(\tau_1\ \mathsf{ref}) & = & \mathcal{R}_O(\tau_1) \\
\mathcal{R}_O(\neg\neg(\tau_1)) & = & \mathcal{R}_O(\tau_1)
\end{array}
$$

The ironing function is defined as an indexed family of functions iron *in Figure 10.5. The double-continuation construction probably is not very useful by itself; but in conjunction with the reference type, it can be used as the type for implementing coroutine-style iterators. That is, the type* τ iter $\triangleq \tau$ cont cont ref *can be used to implement an iterator over an inductive data structure such as trees, which enumerates elements of type* τ *upon "ironing". We leave the detail of implementing such iterators to the Example 10.10.*

The ML implementation is shown in Figure 10.6. As an example, the program

```
let val v1 = ref [3,4]
    and v2 = ref [5,6]
in
```

$$
\begin{array}{lcl}
\mathsf{iron} & : & \forall \tau \in F^{\mathsf{iron}}.\mathcal{R}_I(\tau) \to \mathcal{R}_O(\tau) \\
\mathsf{iron}_{\mathsf{int}}\, x & = & x \\
\mathsf{iron}_{\tau_1\ \mathsf{list}}[x_1, \ldots, x_n] & = & [\mathsf{iron}_{\tau_1} x_1, \ldots, \mathsf{iron}_{\tau_1} x_n] \\
\mathsf{iron}_{\tau_1 \times \tau_2}(x_1, x_2) & = & (\mathsf{iron}_{\tau_1} x_1, \mathsf{iron}_{\tau_2} x_2) \\
\mathsf{iron}_{\tau_1\ \mathsf{ref}}\, x & = & \mathsf{iron}_{\tau_1}(!x) \\
\mathsf{iron}_{\neg\neg(\tau_1)} c & = & \mathsf{iron}_{\tau_1}(\mathsf{callcc}(\lambda k.\mathsf{throw}\, c\, k))
\end{array}
$$

Figure 10.5: The indexed family of ironing functions

```
val Int: int -> int                                              (* ⟨|int|⟩ *)
    = fn x => x
fun List (T: 'a -> 'b): ('a list -> 'b list)                     (* ⟨|list|⟩ *)
    = fn l => map T l
fun Pair (T1: 'a1 -> 'b1) (T2: 'a2 -> 'b2)                       (* ⟨| × |⟩ *)
    : ('a1 * 'a2) -> ('b1 * 'b2)
    = fn (x1, x2) => (T1 x1, T2 x2)
fun Ref (T: 'a -> 'b): ('a ref -> 'b)                            (* ⟨|ref|⟩ *)
    = fn cell => T (! cell)
fun DblNeg (T: 'a -> 'b): ('a cont cont -> 'b)                   (* ⟨|¬¬|⟩ *)
    = fn (c: 'a cont cont) =>
         T (callcc (fn (k: 'a cont) => throw c k))
fun Iter (T: 'a -> 'b): ('a cont cont ref -> 'b)                 (* ⟨|iter|⟩ *)
    = Ref (DblNeg T)

fun iron T = T                                                   (* f_iron *)
```

Figure 10.6: iron in ML: ad-hoc encoding

```
        iron (List (Ref (List Int))) [v1, v2, v1, v2]
    end
```

evaluates to `[[3, 4], [5, 6], [3, 4], [5, 6]]`.

Example 10.10 (Coroutine-style iterators). *A double continuation type, say* τ cont cont*, can be understood as the type of one-shoot* τ*-typed value producers. To invoke such a producer, i.e., to retrieve the* τ*-typed value, the caller should 'throw' to the producer, which is a continuation, its own current continuation, which is of type* τ cont*. This is realized through the definition of* $\mathsf{iron}_{\neg\neg(\tau)}$ *in Figure 10.5.*

If we further make the double continuation type mutable through a reference cell, i.e., employ the type τ cont cont ref*, we can implement a producer thread by changing the stored continuation when suspending the thread. The structure* `Iterator` *Figure 10.7 realizes this idea. In particular, the function* `makeIterator` *convert a "producer" function to an iterator of the mutable double continuation type. This producer function takes as arguments a function to yield the control of the thread and "produce" an answer. The function* `makeIterator` *takes also a second argument to provide a default value, which the iterator should produce after the return from the "producer" function.*

Figure 10.8 (page 112) presents an example of using structure `Iterator`*; it turns a tree to such an iterator. As an example, the program*

```
val iter1
  = tree2Iter (ND(ND(ND(LF 1, LF 2), LF 3), ND (LF 4, ND(LF 5, LF 6)))) ~1
val iter2 = tree2Iter (ND(ND(LF 5, LF 4), ND(LF 3, ND(LF 2, LF 1)))) ~2
val pI    = [iter1, iter2]
val zipped
  = iron (List (List (Iron.Iter Int))) [pI, pI, pI, pI, pI, pI, pI, pI]
```

zips together the traversals of the two trees to produce the following result.

```
structure Iterator =
struct
  local open SMLofNJ.Cont in
  type 'a iterator = 'a cont cont ref
  fun makeIterator (producer: ('a -> unit) -> unit) (dflt: 'a)
    : 'a cont cont ref
    = let val stream = ref (dblNeg dflt)
          val _ =
            callcc(fn back: unit cont =>
              let val consumer_k_ref =
                    ref (callcc(fn init_c =>
                          (stream := init_c; throw back ())))
              in
                (producer (fn v =>
                            (consumer_k_ref :=
                              callcc(fn (c: 'a cont cont) =>
                                ((stream := c);
                                (throw (!consumer_k_ref) v))))));
                                (stream := dblNeg dflt);
                                (throw (!consumer_k_ref) dflt)
              end)
      in  stream
      end
  end
end
```

Figure 10.7: Iterators from mutable double continuation

```
[[1,5],[2,4],[3,3],[4,2],[5,1],[6,~2],[~1,~2],[~1,~2]]
```

10.6 Assessment of the approach

The ad hoc encoding of a type used in the previous sections is exactly the specialized value in the indexed family at the particular type index. There are several advantages to this approach:

- Type safety is automatically ensured by the ML type system: case-analysis on types, though it appears in the formulation, does not really occur during

```
structure BTIterator =
struct
  structure I = Iterator
  datatype 'a BTree = ND of 'a BTree * 'a BTree | LF of 'a
  fun tree2Iterator (T: 'a BTree) (dflt: 'a): 'a I.iterator
    = I.makeIterator
        (fn (produce: 'a -> unit) =>
          let fun traverse (ND(ltree, rtree))
                = ((traverse ltree); (traverse rtree))
                | traverse (LF(n))
                = produce n
          in  traverse T
          end)
        dflt
end
```

Figure 10.8: Iterators for binary trees

program execution. For a type-indexed family of values $v : \forall\beta \in F.Q$, the encoding $\langle\!|\tau|\!\rangle = v_\tau$ of a particular type index τ already has the required type $T_\tau = Q\{\tau/\beta\}$. Often, the value v_τ is a function that takes some argument whose type depends on type τ. Since the specific type of this argument is manifested in the type T_τ, input arguments of illegal types are rejected.

- In some other approaches that do not make the type argument explicit (e.g., using classes of an object-oriented language), one would need to perform case-analysis on tagged values (including dynamic dispatching), which would require the type index to appear at the input position. In our approach, however, the type index τ could appear at any arbitrary position in type T_τ; this has been used, for example, in the implementation of the $\uparrow$ functions for type-directed partial evaluation.

But this simple solution has a major drawback: the loss of composability. One should be able to decompose the task of writing a large type-indexed function into writing several smaller type-indexed functions and then combining them. This would require that the encoding of a type be sharable by these different functions, each of which uses the encoding to obtain the specific value indexed at this type, but from different indexed families. However, the above simple solution of interpreting every type directly as the specific value would result in each type-indexed function having a different set of interpretations of type constructors, thereby disallowing sharing of the type encodings.

Consider the following toy example: on the family $F^{\mathsf{int},\mathsf{list}}$ of types, we define yet another type-indexed function $\mathsf{super\_reverse}$, which recursively reverses a list at each level.

Example 10.11 (Super reverse). *Let us consider the type-indexed family of functions* $\mathsf{super\_reverse}$*, defined as follows.*

$$\begin{aligned}
\mathsf{super_reverse} &: \forall\alpha \in F^{\mathsf{int},\mathsf{list}}.\alpha \to \alpha \\
\mathsf{super_reverse}_{\mathsf{int}} &= \lambda x.x \\
\mathsf{super_reverse}_{\tau_1\ \mathsf{list}} &= \lambda[x_1,\ldots,x_n].[\mathsf{super_reverse}_{\tau_1} x_n,\ldots,\mathsf{super_reverse}_{\tau_1} x_1]
\end{aligned}$$

By interpreting the types, we obtain the ML implementation in Figure 10.9.

```
fun Int x = x                                          (* ⟨|int|⟩ *)
fun List T = rev o (map T)                             (* ⟨|list|⟩ *)
fun super_reverse T = T                                (* f_super_reverse *)
```

Figure 10.9: $\mathsf{super\_reverse}$ in ML: ad-hoc encoding

Each of function flatten and function super_reverse can be used separately, but we cannot write an expression such as

```
fn T => (flatten T) o (super_reverse T)
```

to use them in combination, i.e., to reverse a list recursively and then flatten the result, because the functions `Int` and `List` are defined differently in the two programs. (Notice that the effect of composing function super_reverse and function flatten amounts to reversing the flattened form of the original list, i.e., flatten $\circ$ super_reverse $=$ super_reverse $\circ$ flatten.)

This problem can be evaded in a *non-modular* fashion, if we know in advance *all* possible families $v, v' \ldots$ of values that are indexed by the same family of (type) indices: we can simply tuple all the values together for the type interpretation. Every function f_{v_i}, then, is defined to project out the appropriate component from the type interpretation. Indeed, our previous program of type-directed partial evaluation (Figure 10.3, page 101) illustrates such a tupling.

Chapter 11

Value-Independent Type Encoding

In this chapter, we develop two approaches to encoding types independently of the type-indexed values defined on them, i.e., we should be able to define the encodings $\langle\!|\tau|\!\rangle$ of a family F of types τ, so that given any value v indexed by this family of types, a function f_v that satisfies Equation (10.2) on page 97 can be constructed. In contrast to the solution in the previous section, which interprets types τ using values v_τ directly and is value-dependent, a value-independent type encoding enables different type-indexed values $v, v', \ldots$ to share a family of type encodings, resulting in more modular programs using type-indexed values. We present the following two approaches to value-independent type encoding:

- as an abstraction of the formulation of a type-indexed value, and
- as a universal interpretation of types as tuples of embedding and projection

functions between verbatim values and tagged values.

11.1 Abstracting type encodings

If the type encoding is value-independent, the function f_v representing type-indexed value v should carry the information of the value constructions e_i in a specification in the form of the set of equations given in (9.2) on page 87. This naturally leads to the following approach to type encoding: a type-indexed value v is characterized as an n-ary tuple $\vec{e} = (e_1, \ldots, e_n)$ of the value constructions, and the value-independent type interpretation $\langle\!|\tau|\!\rangle$ maps this specification to the specific value v_τ.

$$\langle\!|\tau|\!\rangle\vec{e} = v_\tau \tag{11.1}$$

With Equation (10.1) on page 96, we require the encoding of type constructors c_i to satisfy

$$\begin{aligned}
& \langle\!|c_i|\!\rangle(\langle\!|\tau_1|\!\rangle, \ldots, \langle\!|\tau_m|\!\rangle)\vec{e} \\
= \;& \langle\!|c_i(\tau_1, \ldots, \tau_m)|\!\rangle\vec{e} && (\textit{by (10.1)}) \\
= \;& v_{c_i(\tau_1,\ldots,\tau_m)} && (\textit{by (11.1)}) \\
= \;& e_i(v_{\tau_1}, \ldots, v_{\tau_m}) && (\textit{by (9.2)}) \\
= \;& e_i(\langle\!|\tau_1|\!\rangle\vec{e}, \ldots, \langle\!|\tau_m|\!\rangle\vec{e}) && (\textit{by (11.1)})
\end{aligned}$$

By this derivation, we have

Theorem 11.1. *The value-independent encodings of type constructors*

$$\langle\!|c_i|\!\rangle = \lambda(x_1, \ldots, x_m).\lambda\vec{e}.e_i(x_1\vec{e}, \ldots, x_m\vec{e})$$

and the function $f_v(x) = x(e_1, \ldots, e_n)$ *implement the corresponding type-indexed value* v.

This approach seems to be readily usable as the basis of programming type-indexed values in ML. However, the restriction of ML type system that universal quantifiers on type variables must appear at the top level again makes this approach impossible. For example, let us try to encode types in the family $F^{\mathsf{exp},\mathsf{func}}$, and use them to program type-directed partial evaluation in ML (Figure 11.1).

```
val Base = fn (base_v, func_v) => base_v
fun T1 --> T2 = fn (spec_v as (base_v, func_v))
                => func_v (T1 spec_v) (T2 spec_v)

fun reify T =
    let val (reify_T, _) =
        T ((fn v => v, fn e => e),       (* base_v *)
                                         (* func_v *)
           fn (reify_T1, reflect_T1) =>
           fn (reify_T2, reflect_T2) =>
              ...          (* (reify_T, reflect_T) *)
         )
    in reify_T end
```

Figure 11.1: An unsuccessful encoding of $F^{\mathsf{exp},\mathsf{func}}$ and TDPE

The definition of `reify` and `reflect` at higher types is as before and omitted here for brevity. This program will not work, because the λ-bound variable `spec_v` can only be used monomorphically in the function body. This forces all uses of `func_v` to have the same monotype; as an example, the type encoding `Base --> (Base --> Base)` causes a type error, because the two uses of variable `func_v` (one being applied, the other being passed to lower type interpretations) have different monotypes.

Indeed, the type of the argument of `reify`, a type encoding $\langle\!|\tau|\!\rangle$ constructed

using `Base` and `-->`, is somewhat involved:

$$
\begin{array}{l}
\langle\!|\tau|\!\rangle \ : \ \forall \mathsf{obj} : * \to *. \\
\qquad \forall \mathsf{base\_type} : *. \\
\qquad (\mathsf{base\_type\ obj} \ \times \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (* \ \mathsf{base\_v} \ *) \\
\qquad\ (\forall \alpha : *, \beta : *.(\alpha\ \mathsf{obj}) \to (\beta\ \mathsf{obj}) \to ((\alpha \to \beta)\ \mathsf{obj}))) \to \quad (* \ \mathsf{func\_v} \ *) \\
\qquad \tau\ \mathsf{obj}
\end{array}
$$

Here, the type constructor obj constructs the type T_τ of the specific value v_τ from a type index τ, and the type base_type gives the base type index. What we need here is *first-class polymorphism*, which allows nested quantified types, as used in the type of argument `func_v`. Substantial work has been done in this direction, such as allowing selective annotations of λ-bound variables with polymorphic types [82] or packaging of these variables using polymorphic datatype components [53]. Moreover, *higher-order polymorphism* [52] is needed to allow parameterizing over a type constructor, e.g., the type constructor obj.

In fact, such type encodings are similar to a Martin-Löf-style encoding of inductive types using the corresponding elimination rules in System F_ω, which does support both first-class polymorphism and higher-order polymorphism in an explicit form [36, 92].

11.2 Explicit first-class and higher-order polymorphism in SML/NJ

The module system of Standard ML provides an explicit form of first-class polymorphism and higher-order polymorphism. Quantifying over a type or a type

constructor is done by specifying the type or type constructor in a signature, and parameterizing functors with this signature. To recast the higher-order functions in Figure 11.1 (page 117) into functors, we also need to use higher-order functors which allows functors to have functor arguments or results. Such higher-order modules are supported by Standard ML of New Jersey [4], which extends Standard ML with higher-order functors [102]. Figures 11.2 and 11.3 (page 121) gives a program for type-directed partial evaluation using higher-order functors.

Here, a `Type` encoding is a functor from a structure with signature `IndValue`, which is a specification of type-indexed values, to a structure with signature `SpecValue`, which denotes a value of the specific type. The type `my_type` gives the particular type index τ, and the type `base_type` and the type constructor `obj` are as described in Section 11.1.

It is however cumbersome and time-consuming to use such functor-based encodings. The following example illustrates how to partially evaluate (residualize) the function $\lambda x.x$ with type $(\mathsf{base} \to \mathsf{base}) \to (\mathsf{base} \to \mathsf{base})$.

```
local structure T   = Arrow(Arrow(Base)(Base))
                            (Arrow(Base)(Base))
      structure v_T = T.F(reify_reflect)
in
      val result = #1(v_T.v) (fn x => x)
end
```

Furthermore, since ML functions cannot take functors as arguments, we must define functors to use such functor-encoded type arguments. Therefore, even though this approach is conceptually simple and gives clean, type-safe and value-

```
signature SpecValue =
sig
  type 'a obj
  type my_type
  val v: my_type obj
end

signature IndValue =
sig
  type 'a obj
  type base_type
  val Base : base_type obj
  val Arrow: 'a obj -> 'b obj -> ('a -> 'b) obj
end

signature Type =
sig
  functor F(Obj: IndValue): SpecValue
    where type 'a obj = 'a Obj.obj
end

structure Base: Type =
struct
  functor F(Obj: IndValue): SpecValue =
    struct
      type 'a obj = 'a Obj.obj
      type my_type = Obj.base_type
      val v = Obj.Base
    end
end

functor Arrow(T1: Type) (T2: Type): Type =
struct
  functor F(Obj: IndValue): SpecValue =
    struct
      type 'a obj = 'a Obj.obj
      structure v_T1 = T1.F(Obj)
      structure v_T2 = T2.F(Obj)
      type my_type = v_T1.my_type -> v_T2.my_type
      val v = Obj.Arrow v_T1.v v_T2.v
    end
end
```

Figure 11.2: Encoding $F^{\mathsf{exp},\mathsf{func}}$ using higher-order functors

```
structure TDPE: IndValue =
  struct
    type 'a obj = ('a -> exp) * (exp -> 'a)
    type base_type = exp
    val Base = (fn v => v, fn e => e)
    fun Arrow (reif1, refl1) (reif2, refl2)
      = (fn (f: 'a -> 'b) =>
            let val x = Gensym.new()
            in  LAM(x, reif2 (f (refl1 (VAR x))))
            end,
         fn (e: exp) =>
            fn (v: 'a) => refl2 (APP(e, reif1 v)))
    end
```

Figure 11.3: Type-directed partial evluation using the functor-based encoding

independent type encodings, the syntactic overhead in using the type system makes the approach somewhat tedious and difficult to be used for programming in ML.

11.3 Embedding/projection functions as type interpretation

The alternative approach to value-independent type encodings is (maybe somewhat surprisingly) based on programming with tagged values of user-defined universal datatypes. Before describing this approach, let us look at how tagged values are often used to program functions with type arguments.

First of all, for a type-indexed value v whose type index τ appears at the position of input arguments, the tags attached to the input arguments are enough to guide the computation. For examples, the tagged-value version of functions `flatten` and `super_reverse` is as follows:

```
datatype tagIntList =
    INT of int
  | LST of tagIntList list

fun flattenTg (INT x)
    = [x]
  | flattenTg (LST l)
    = foldr (op @) [] (map (fn x => flattenTg x) l)
fun super_reverseTg (INT v)
    = INT v
  | super_reverseTg (LST l)
    = LST (rev (map super_reverseTg l))
```

In more general cases, if the type index τ can appear at any position of the type T_τ of specific values v_τ, then a description of type τ using a datatype must be provided as a function argument.

However, this approach suffers from several drawbacks:

1. Verbatim values cannot be directly used.

2. If an explicit encoding of a type τ is provided, one cannot ensure at compile time its consistency with other input arguments whose types depend on type τ; in other words, run-time 'type-errors' can happen due to unmatched tags.

Can we avoid these problems while still using universal datatypes? To solve the first problem, we want the program to automatically tag a verbatim value according to the type argument. To solve the second problem, if all tagged values

are generated from verbatim values under the guidance of type arguments, then they are guaranteed to conform to the type encoding, and run-time 'type-errors' can be avoided.

The automatic tagging process that embeds values of various types into values of a universal datatype is called an *embedding* function. Its inverse process, which removes tags and returns values of various types, is called a *projection* function. Interestingly, these functions are type-indexed themselves, thus they can be programmed using the ad-hoc method described in Chapter 10. Using the embedding function and projection function of a type τ as its encoding gives another value-independent type encoding method for type-indexed values.

For each family T of types τ inductively defined in the form of Equation (9.1) on page 86, we first define a datatype U of tagged values, as well as a datatype *typeExp*U (type expression) to represent the type structure. Next, we use the following interpretation as the type encoding:

$$\begin{array}{rcll} \langle\!|\tau|\!\rangle & = & \langle emb_\tau, proj_\tau, tE_\tau \rangle & \\ emb_\tau & : & \tau \to U & \text{(embedding function)} \\ proj_\tau & : & U \to \tau & \text{(projection function)} \\ tE_\tau & : & typeExp & \text{(type expression)} \end{array} \tag{11.2}$$

Finally, we use the embedding and projection functions as basic coercions to convert a value based on a universal datatype to the type of the specific value v_τ.

We continue to illustrate the approach in Section 11.3.1, and then formally present the general approach in Section 11.3.2.

11.3.1 Examples

Taking the family $F^{\mathsf{int},\mathsf{list}}$ of types, we can encode the type constructors as:

```
datatype typeExpL = tInt | tLst of typeExpL
val Int = (fn x => INT x, fn (INT x) => x, tInt)
fun List (T as (emb_T, proj_T, tE_T)) =
    (fn l => LST (map emb_T l),
     fn LST l => map proj_T l,
     tLst tE_T)
```

and then the functions `flatten` and `super_reverse` are defined as

```
fun flatten (T as (emb, _, _)) v = flattenTg (emb v)
fun super_reverse (T as (emb, proj, _)) v =
        proj (super_reverseTg (emb v))
```

Now that the type encoding is neutral to different type-indexed values, they can be combined to share the same type argument. For example, the function

```
fn T => (flatten T) o (super_reverse T)
```

defines a type-indexed function that composes `flatten` and `super_reverse`.

The other component of the interpretation, the type expression `tE`, is used for those functions where the type indices do not appear at the input argument positions, such as the `reflect` function. In these cases, a tagged-value version of the type-indexed value need to perform case analysis on the type expression `tE`. As an example, the code of type-directed partial evaluation using this new type interpretation is presented in Figure 11.4.

```
datatype 'base tagBaseFunc =
    BASE of 'base
  | FUNC of ('base tagBaseFunc) -> ('base tagBaseFunc)
datatype typeExpF =
    tBASE
  | tFUNC of typeExpF * typeExpF

infixr 5 -->

val Base = (fn x => (BASE x), fn (BASE x) => x, tBASE)
fun ((T1 as (I_T1, P_T1, tE1)) -->
     (T2 as (I_T2, P_T2, tE2))) =
    (fn f => FUNC (fn tag_x => I_T2 (f (P_T1 tag_x))),
     fn FUNC f => (fn x => P_T2 (f (I_T1 x))),
     tFUNC(tE1,tE2))

val rec reifyTg =
  fn (tBASE, BASE v) => v
   | (tFUNC(tE1,tE2), FUNC v) =>
       let val x1 = Gensym.fresh "x" in
         LAM(x1, reifyTg
                 (tE2, v (reflectTg (tE1, (VAR x1)))))
       end
and reflectTg =
  fn (tBASE, e) => BASE(e)
   | (tFUNC(tE1,tE2), e) =>
       FUNC(fn v1 => reflectTg
                     (tE2, APP (e, reifyTg (tE1, v1))))

fun reify (T as (emb, _, tE)) v = reifyTg(tE, emb v)
```

Figure 11.4: Embedding/projection-based encoding for TDPE

Recall that the definition of the functions `reifyTg` and `reflectTg` will cause matching-inexhaustive compilation warnings, and invoking them might cause run-time exceptions. In contrast, function `reify` is safe in the sense that if the argument `v` type-checks with the domain type of the embedding function `emb`, then the resulting tagged expression must comply with the type expression `tE`.

This value-independent type encoding can be used for the form of 'type specialization' proposed by Danvy [14], where the partial evaluator and the projection function are type-indexed by the same family of types.

11.3.2 Universality

In this section, we argue that the above approach based on embedding and projection functions indeed provides a value-independent encoding for a majority of type constructors. The idea is that the embedding/projection encoding forms a *universal* type-indexed family of values, in that any other family of values indexed by the same family can be constructed from this particular family.

We assume the following conditions about the types:

1. All the type constructions c_i (in Equations 9.2, page 87) build a type only from component types covariantly and/or contravariantly. The constructed type can use the same component type both covariantly and contravariantly at its different occurrences, as in the example of type-directed partial evaluation. This condition rules out, for example, the reference type constructor.

2. The type Q is covariant or contravariant in every occurrence of the type variable τ.

Universal Type and Embedding/projection Pairs We first define an ML datatype U to distinctively represent values of different types in the type family F. This is done by tagging all the branches of type constructions c_i. Without loss of generality, we assume no other type variable freely occurring in the type

constructions c_i. Type variables can be dealt with by parameterization over type U.

$$\begin{array}{rcl} \textbf{datatype } U & = & tag_{c_1} \textbf{ of } c_1(\overbrace{U, \ldots, U}^{m_1}) \\ & \vdots & \\ & | & tag_{c_n} \textbf{ of } c_n(\underbrace{U, \ldots, U}_{m_n}) \end{array}$$

We also define a datatype *typeExpU* to describe the structure of a particular type in the type family F:

$$\begin{array}{rcl} \textbf{datatype } \mathit{typeExpU} & = & \mathit{tEc}_1 \textbf{ of } (\mathit{typeExpU})^{m_1} \\ & \vdots & \\ & | & \mathit{tEc}_n \textbf{ of } (\mathit{typeExpU})^{m_n} \end{array}$$

Condition 1 ensures the existence of the embedding/projection pairs between every inductively constructed type index $\tau \in F$ and the universal type U. Such pairs witness the apparent isomorphisms between the values of type τ, denoted as $\mathit{Val}(\tau)$, and the corresponding set of tagged values of type U, denoted as $\mathit{UVal}(\tau)$.

To see that such pairs always exist, consider the category **Type** whose objects are types and whose morphisms are coercions (see section 2.1 of Henglein's article [45]). Every type construction c_i is interpreted as a multi-functor that is either covariant or contravariant in each of its argument:

$$C_i : \mathbf{Type}^{p_1} \times \mathbf{Type}^{p_2} \times \cdots \times \mathbf{Type}^{p_{m_i}} \to \mathbf{Type}$$

Here p_j's are the polarities of the arguments: if the type construction c_i is covariant in its j-th argument, then $p_j = +$ and $\mathbf{Type}^{p_i} = \mathbf{Type}$; if it is contravariant

in its j-th argument, then $p_j = -$ and $\mathbf{Type}^{p_j} = \mathbf{Type}^{op}$. Given pairs of embedding $emb_{\tau_j} : \tau_j \rightsquigarrow U$ and projection $proj_{\tau_j} : U \rightsquigarrow \tau_j$ at the component type, we can apply the functor C_i to induce another pair of coercions for the constructed type $\tau = c_i(\tau_1, \ldots, \tau_{m_i})$.

$$\begin{array}{rcl} \varepsilon_\tau & = & C_i(emb_{\tau_1}^{p_1}, \ldots, emb_{\tau_{m_i}}^{p_{m_i}}) \quad : \quad \tau \rightsquigarrow c_i(\overbrace{U, \ldots, U}^{m_i}) \\ \pi_\tau & = & C_i(proj_{\tau_1}^{p_1}, \ldots, proj_{\tau_{m_i}}^{p_{m_i}}) \quad : \quad c_i(\underbrace{U, \ldots, U}_{m_i}) \rightsquigarrow \tau \end{array}$$

Here, $emb_\tau^+ = emb_\tau$, $emb_\tau^- = proj_\tau$, $proj_\tau^+ = proj_\tau$ and $proj_\tau^- = emb_\tau$ for all types τ. Composing coercions ε_τ and π_τ with the tagging and untagging operations for the tag tag_{c_i} respectively gives the embedding/projection pair at the type τ. By structural induction, all types τ in the type family T have the embedding/projection pairs.

Let us make concrete the above construction for the type constructions in ML. For Condition 1, we assume that every type construction c_i is inductively generated using the ML type constructions in Figure 11.5.

Lemma 11.2. *Let c be a m-ary type construction generated by the grammar in Figure 11.5. Given types τ_j (for $j = 1, \ldots, m$), and for each type a pair of embedding $emb_j : \tau_j \rightsquigarrow U$ and projection $proj_j : U \rightsquigarrow \tau_j$ (defined on $UVal(\tau_j)$), which are inverse to each other between $Val\tau_j$ and $UVal\tau_j$, one can induce a pair of functions $\varepsilon_\tau : \tau \rightsquigarrow c(U, \ldots, U)$ and $\pi_\tau : c(U, \ldots, U) \rightsquigarrow \tau$ where $\tau = c(\tau_1, \ldots, \tau_m)$, which are inverse to each other between the set $Val\tau$ and $UVal\tau$.*

Proof. By structural induction on type $\tau = c(\tau_1, \ldots, \tau_m)$.

$c(\tau_1, \ldots, \tau_m) = \tau_j$. Define $\varepsilon_\tau = emb_j : \tau_j \rightsquigarrow U$ and $\pi_\tau = proj_j : U \rightsquigarrow \tau_j$. They

$$
\begin{array}{lcll}
c(\tau_1, \ldots, \tau_m) & = & \tau_j \quad (j = 1, \ldots, m) & (\textit{Type argument}) \\
& | & \beta & (\textit{Free type variable}) \\
& | & A & (\textit{Built-in atomic types}) \\
& | & c_1(\tau_1, \ldots, \tau_m) \to c_2(\tau_1, \ldots, \tau_m) & (\textit{Function}) \\
& | & c_1(\tau_1, \ldots, \tau_m)\ \mathsf{list} & (\textit{List}) \\
& | & c_1(\tau_1, \ldots, \tau_m) * \cdots * c_l(\tau_1, \ldots, \tau_m) & (\textit{Tuple of length } l)
\end{array}
$$

Figure 11.5: Formation of a type construction c

are inverse to each other by the condition of the lemma.

$c(\tau_1, \ldots, \tau_m) = \beta$. Here β is a freely occurring type variable, thus $c(U, \ldots, U) = \beta$. Define $\varepsilon_\tau = \pi_\tau = \lambda x.x : \beta \rightsquigarrow \beta$, which are inverse to each other.

$c(\tau_1, \ldots, \tau_m) = A$. Since $c(U, \ldots, U) = A$, setting $\varepsilon_\tau = \pi_\tau = \lambda x.x : A$ gives the pair.

$c(\tau_1, \ldots, \tau_m) = c_1(\tau_1, \ldots, \tau_m) \to c_2(\tau_1, \ldots, \tau_m)$. By the induction hypotheses, we have $\varepsilon_{c_1(\tau_1,\ldots,\tau_m)} : c_1(\tau_1, \ldots, \tau_m) \rightsquigarrow c_1(U, \ldots, U)$ and its inverse $\pi_{c_1(\tau_1,\ldots,\tau_m)} : c_1(U, \ldots, U) \rightsquigarrow c_1(\tau_1, \ldots, \tau_m)$, together with $\varepsilon_{c_2(\tau_1,\ldots,\tau_m)} : c_2(\tau_1, \ldots, \tau_m) \rightsquigarrow c_2(U, \ldots, U)$ and its inverse $\pi_{c_2(\tau_1,\ldots,\tau_m)} : c_2(U, \ldots, U) \rightsquigarrow c_2(\tau_1, \ldots, \tau_m)$. Now, define

$$
\begin{aligned}
\varepsilon_\tau f &= \varepsilon_{c_2(\tau_1,\ldots,\tau_m)} \circ f \circ \pi_{c_1(\tau_1,\ldots,\tau_m)} \\
\pi_\tau f &= \pi_{c_2(\tau_1,\ldots,\tau_m)} \circ f \circ \varepsilon_{c_1(\tau_1,\ldots,\tau_m)}
\end{aligned}
$$

It is easy to verify that these two functions have the required types, and

they are inverse to each other.

$c(\tau_1, \ldots, \tau_m) = c_1(\tau_1, \ldots, \tau_m)$ **list.** By the induction hypothesis, we have the embedding $\varepsilon_{c_1(\tau_1,\ldots,\tau_m)} : c_1(\tau_1, \ldots, \tau_m) \rightsquigarrow c_1(U, \ldots, U)$ and as its inverse the projection $\pi_{c_1(\tau_1,\ldots,\tau_m)} : c_1(U, \ldots, U) \rightsquigarrow c_1(\tau_1, \ldots, \tau_m)$. Now, let

$$\begin{aligned} \varepsilon_\tau L &= \mathbf{map}\ L\ \varepsilon_{c_1(\tau_1,\ldots,\tau_m)} \\ \pi_\tau L &= \mathbf{map}\ L\ \pi_{c_1(\tau_1,\ldots,\tau_m)} \end{aligned}$$

It is easy to verify that these two functions have the required types, and they are inverse to each other.

$c(\tau_1, \ldots, \tau_m) = c_1(\tau_1, \ldots, \tau_m) * \cdots * c_l(\tau_1, \ldots, \tau_m)$**.** By induction hypothesis, we have embeddings $\varepsilon_{c_i(\tau_1,\ldots,\tau_m)}$ and projections $\pi_{c_i(\tau_1,\ldots,\tau_m)}$, which are pairwise inverse. Now, let

$$\begin{aligned} \varepsilon_\tau(x_1, \ldots, x_l) &= (\varepsilon_{c_1(\tau_1,\ldots,\tau_m)}x_1, \ldots, \varepsilon_{c_l(\tau_1,\ldots,\tau_m)}x_l) \\ \pi_\tau(x_1, \ldots, x_l) &= (\pi_{c_1(\tau_1,\ldots,\tau_m)}x_1, \ldots, \pi_{c_l(\tau_1,\ldots,\tau_m)}x_l) \end{aligned}$$

It is easy to verify that these two functions have the required types, and they are inverse to each other.

□

Theorem 11.3. *For all types $\tau \in F$, there is a pair of embedding $emb_\tau : \tau \rightsquigarrow U$ and projection $proj_\tau : U \rightsquigarrow \tau$ which are inverse to each other between $Val\tau$ and $UVal\tau$.*

Proof. By induction on type τ.

$\tau = c_i(\tau_1, \ldots, \tau_{m_i})$. The induction hypotheses for every type τ_j where $1 \leq j \leq m_i$ says that $emb_{\tau_j} : \tau_j \rightsquigarrow U$ and $proj_{\tau_j} : U \rightsquigarrow \tau_j$ exist and are inverse to each other. By Lemma 11.2, we can induce a pair of inverse functions $\varepsilon_\tau : \tau \rightsquigarrow c_i(U, \ldots, U)$ and $\pi_\tau : c_i(U, \ldots, U) \rightsquigarrow \tau$. Define

$$\begin{aligned} emb_\tau &: \tau \rightsquigarrow U \\ emb_\tau(x) &= tag_{c_i}(\varepsilon_\tau(x)) \\ proj_\tau &: U \rightsquigarrow \tau \\ proj_\tau(tag_{c_i}(x)) &= \pi_\tau(x) \end{aligned}$$

It is easy to verify that the two functions are inverse to each other.

□

The proof above essentially gives an algorithm for computing the embedding / projection pair for every type τ. Notice this algorithm itself is specified in a type-indexed form, that the pair for a constructed type is computed from the pairs for its component types; therefore, we can use the ad-hoc approach to program the type interpretation in the form of Equation (11.2).

Embedding/Projection for the result type The type encoding $\langle\!|\tau|\!\rangle$ of a type $\tau \in F$ gives the pairs of embedding and projection between this type and the universal type U. Now, for a type-indexed family v of values with the type scheme $\forall\alpha \in F.Q$, we need to compute the embedding and the projection between type $Q\{\tau/\alpha\}$ and type $Q\{U/\alpha\}$ for any given type $\tau \in F$ from its type encoding $\langle\!|\tau|\!\rangle$. This makes it possible to first compute the universal version of the value v_τ, which is of type $Q\{U/\alpha\}$, and then project it to the specific type

$Q\{\tau'/\alpha\}$. Condition 2 (page 126) ensures the existence of these embedding/ projection pairs.

Like before, Condition 2 can be made concrete in terms of ML type constructions. We assume that the type Q is inductively generated using the ML type constructions in Figure 11.6.

$$
\begin{array}{rcll}
Q & = & \tau & (\textit{indexed type } \tau) \\
 & | & A & (\textit{Built-in atomic types}) \\
 & | & Q_1 \to Q_2 & (\textit{Function}) \\
 & | & Q_1 \ \mathsf{list} & (\textit{List}) \\
 & | & Q_1 \times \cdots \times Q_l & (\textit{Tuple of length } l)
\end{array}
$$

Figure 11.6: Formation of the type Q of a type-indexed value

Theorem 11.4. *Let Q be a type with free type variable α generated by the grammar in Figure 11.6. Given a type τ and the pair of inverse functions $emb_\tau : \tau \rightsquigarrow U$ and $proj_\tau : U \rightsquigarrow \tau$, one can induce a pair of functions $e^Q_\tau : Q\{\tau/\alpha\} \rightsquigarrow Q\{U/\alpha\}$ and $p^Q_\tau : Q\{U/\alpha\} \rightsquigarrow Q\{\tau/\alpha\}$ which are inverse to each other.*

Proof. By induction on type Q. The proof is similar to the previous ones; for brevity, here we simply gives the construction of e^Q_τ and p^Q_τ. It is straightforward to verify that each of them is a pair of inverse functions.

$Q = \tau$**.** Define $e^Q_{\tau'} = emb_{\tau'}$ and $p^Q_{\tau'} = proj_{\tau'}$.

$Q = A$**.** Define $e^Q_{\tau'} = p^Q_{\tau'} = \lambda x.x : A \to A$.

$Q = Q_1 \to Q_2$**.** Define $e^Q_{\tau'}\ f = e^{Q_1}_{\tau'} \circ f \circ p^{Q_2}_{\tau'}$ and $p^Q_{\tau'}\ f = p^{Q_1}_{\tau'} \circ f \circ e^{Q_2}_{\tau'}$.

$Q = Q_1$ list**.** Define $e^Q_{\tau'}\ L = \mathbf{map}\ L\ e^{Q_1}_{\tau'}$ and $p^Q_{\tau'}\ L = \mathbf{map}\ L\ p^{Q_1}_{\tau'}$.

$Q = Q_1 * \cdots * Q_l$**.** Define $e^Q_{\tau'}\ (x_1, \ldots, x_l) = (e^{Q_1}_{\tau'}\ x_1, \ldots, e^{Q_l}_{\tau'}\ x_l)$
and $p^Q_{\tau'}\ (x_1, \ldots, x_l) = (p^{Q_1}_{\tau'}\ x_1, \ldots, p^{Q_l}_{\tau'}\ x_l)$.

□

In fact, the proof itself provides an algorithm for computing e^Q_τ and p^Q_τ for a fixed type Q from the embedding/projection pair of type τ, which is included in the "universal" encoding encoding $\langle\!|\tau|\!\rangle$.

Type-indexed values from universal values Now, we can write a function $f^U_v : \mathit{typeExp}U \to Q\{U/\alpha\}$, the universal-datatype version of the type-indexed value v, such that for each type $\tau \in T$, the value $f^U_v(tE_\tau)$ is equivalent to the verbatim value v_τ embedded into the universal type $Q\{U/\alpha\}$.

$$f^U_v(tE_\tau) = e^Q_\tau(v_\tau)$$

It is then sufficient to define the function f_v as

$$f_v\langle \mathit{emb}_\tau, \mathit{proj}_\tau, tE_\tau\rangle = p^Q_\tau(f^U_v(tE_\tau)) \tag{11.3}$$

where p^Q_τ is constructed from emb_τ and proj_τ by Theorem 11.4. Function f_v and the universal encoding $\langle\!|\cdot|\!\rangle$ does implement the type-indexed family v (Definition 10.2, page 97); this follows from the fact that $p^Q_\tau \circ e^Q_\tau$ is the identity function, also by Theorem 11.4.

Function f_v^U can be induced from the specification in the form of Equation (9.2) as follows:

$$\begin{aligned} f_v^U(tEc_1(tE_{\tau_1},\ldots,tE_{\tau_{m_1}})) &= e_1^U(f_v^U(tE_{\tau_1}),\ldots,f_v^U(tE_{\tau_{m_1}})) \\ &\vdots \end{aligned}$$

where $e_i^U : (Q\{U/\alpha\})^{m_i} \to Q\{U/\alpha\}$ is a properly instrumented version of $e_i : \forall\alpha_1,\ldots,\alpha_{m_i}.(Q\{\alpha_1/\alpha\}\times\cdots\times Q\{\alpha_{m_i}/\alpha\})\to Q\{c_i(\alpha_1,\cdots,\alpha_{m_i})/\alpha\}$ by adding tagging and untagging operations. This process can be purely mechanical: instantiating all the type variables α_i to type U, and then applying a coercion function induced from the data constructor tag_{c_i} of type U.

Note that the case analysis on the types in Equation (9.2) has been turned into case analysis on their value representations, which are of the uniform type *typeExp*U. This way, the program f_v^U fits into the Hindley-Milner type system.

The following theorem summarizes the approach based on the above construction.

Theorem 11.5. *Encoding types as embedding/projection functions gives a value-independent type encoding for a type family F. Every F-indexed family v of values is implemented by the function f_v defined in Equation (11.3) and the type encoding.*

11.3.3 Comments

The new approach to value-independent type encodings is general and practical. Though this approach is based on universal datatype solutions using tagged values, it overcomes the two original problems of directly using universal datatypes:

- Though the universal datatype version of the indexed value is not type-safe, the coerced value is type-safe in general. This is because verbatim input arguments of various types are mapped into the universal datatype by the embedding function, whose type acts as a filter of input types. Unmatched tags are prevented this way.

- Users do not need to tag the input and/or untag the output; this is done automatically by the program f_v using the embedding and projection functions. From another perspective, this provides a method of *external tagging* using the type structure. While internal tagging incurs a syntactic overhead proportional to the size of the term, external tagging incurs a syntactic overhead proportional to only the size of the type.

This approach is not as efficient as the ad-hoc, value-dependent approach, due to the lengthy tagging and untagging operations and the introduction of extra intermediate data structures. This problem can be overcome using program-transformation techniques such as partial evaluation [57], by specializing the general functions with respect to certain type encodings at compile time, and removing all the tagging/untagging operations. In particular, Danvy showed how it can be naturally combined with type-directed partial evaluation to get a 2-level embedding/projection function [14].

Sometimes, one need to use more involved type constructions that are beyond the reach of Conditions 1 and 2 (page 126), such as recursive type constructions. Here we do not offer a general solution. We hope, though, to use Henglein and Rehof's dynamic-typing calculus [45, 46] to get a general treatment for these

cases.

11.4 Multiple Type Indices

Though our previous examples only demonstrate type-indexed values which have only one type index, the embedding/projection-based approach can be readily applied to implementing values indexed by more than one type index. Figure 11.7 presents an example: an ML function that performs subtype coercion [72]. Given a from-type, a to-type, a list of subtype coercions at base types, and a value of the from-type, this function coerces the value to the to-type and returns it.

Following the general pattern, we first write a function `univ_coerce`, which performs the coercions on tagged values. The function `coerce` then wraps up function `univ_coerce`, by embedding the input argument and projecting the output. For brevity, we have omitted the obvious definition of the related datatypes, and the type interpretations as embedding/projection functions and type expressions of `Int`, `Str`, `List`, `-->`, `**`, some of which have already appeared in previous examples.

The example below builds a subtype coercion `C` : $\mathsf{str} \to \mathsf{str} \rightsquigarrow \mathsf{int} \to \mathsf{str}$, given a base coercion $\mathsf{int} \rightsquigarrow \mathsf{str}$, so that, e.g., the expression `C (fn x => x ^ x) 123` evaluates to `"123123"`.

```
val C = coerce [(tINT, tSTR,
                 fn (INT x) => STR (Int.toString x))]
               (Str --> Str) (Int --> Str)
```

Again, this approach can be combined with type-directed partial evaluation

```
exception nonSubtype of typeExp * typeExp

fun lookup_coerce [] tE1 tE2 = raise nonSubtype(tE1, tE2)
  | lookup_coerce ((t, t', t2t')::Others) tE1 tE2 =
    if t = tE1 andalso t' = tE2 then
        t2t'
    else
        lookup_coerce Others tE1 tE2

fun univ_coerce cl (tFUN(tE1_T1, tE2_T1))
                   (tFUN(tE1_T2, tE2_T2)) (FUN v) =
    FUN (fn x => univ_coerce cl tE2_T1 tE2_T2
         (v (univ_coerce cl tE1_T2 tE1_T1 x)))
  | univ_coerce cl (tLST tE_T1) (tLST tE_T2) (LST v) =
    LST (map (univ_coerce cl tE_T1 tE_T2) v)
  | univ_coerce cl (tPR(tE1_T1, tE2_T1))
                   (tPR(tE1_T2, tE2_T2)) (PR (x, y)) =
    PR (univ_coerce cl tE1_T1 tE1_T2 x,
        univ_coerce cl tE2_T1 tE2_T2 y)
  | univ_coerce cl x y v =
    if x = y then
        v
    else
        (lookup_coerce cl x y) v

fun coerce cl (T1 as (emb_T1, proj_T1, tE_T1))
              (T2 as (emb_T2, proj_T2, tE_T2)) v =
    proj_T2 (univ_coerce cl tE_T1 tE_T2 (emb_T1 v))
```

Figure 11.7: Type-safe coercion function

to obtain 2-level functions, as done by Danvy for coercion functions and by Vestergaard for "à la Kennedy" conversion functions [61, 103].

Chapter 12

Related work: using more expressive type systems

The problem of programming type-indexed values in a statically typed language like ML motivated several earlier works that introduce new features to the type systems. In the following sections, we briefly go through some of these frameworks that provide solutions to type-indexed values.

12.1 Dynamic typing

Realizing that static typing is too restrictive in some cases, there is a line of work on adding dynamic typing [1, 2] to languages with static type systems. Such an approach introduces a universal type `Dynamic` along with two operations for constructing values of type `Dynamic` and inspecting the type tag attached to these values. A dynamic typing approach extends user-defined datatypes in several

ways: the set of type constructions does not need to be known in advance—the type `Dynamic` is extensible; it also allows polymorphism in the represented data. Processing dynamic values is however similar to processing tagged values of user-defined type—both require operations that wrap values and case analysis that removes the wrapping.

A recent approach along the line of dynamic typing, *staged type inference* [93] proposes to defer the type inference of some expressions until run-time when all related information is available. In particular, this approach is naturally combined with the framework of staged computation [27, 100] to support type-safe code generation at run-time. Staged programming helped to solve some of the original problems of dynamic typing, especially those concerning the ease of use.

However, the way type errors are prevented at run-time is to require users to provide 'default values' that have expected types of expressions whose actual types are inferred at run-time; when type inference fails, or the inferred type does not match the context, the default values are used. This is effectively equivalent to providing default exception handlers for run-time exceptions resulting from type inference. The approach is still a dynamic-typing approach, so that the benefit of static debugging offered by a static typing system is lost. For example, the formatting function in [93] will simply return an error when field specifiers do not match the function arguments. On the other hand, it is also because of this possibility of run-time 'type error' that dynamic typing disciplines provide extra flexibility, as shown in applications such as meta-programming and high-level data/code transfer in distributed programming.

12.2 Intensional type analysis

Intensional type analysis [42] directly supports type-indexed values in the language λ_i^{ML} in order to compile polymorphism into efficient unboxed representations. The language λ_i^{ML} extends a predicative variant of Girard's System F_ω with primitives for intensional type analysis, by providing facilities to define constructors and terms by structural induction on monotypes. However, the language λ_i^{ML} is explicitly polymorphic, requiring pervasive type annotations throughout the program and thus making it inconvenient to directly program in this language. Not surprisingly, the language λ_i^{ML} is mainly used as a typed intermediate language.

12.3 Haskell type classes

The *type-class* mechanism in Haskell [41] also makes it easy to program type-indexed family of values: the declaration of a type class should include all the type-indexed value needed, and every value construction e_i should be implemented as an instance declaration for the constructed type, assuming the component types are already instances of the type class. One way of implementing type classes is to translate the use of type classes to arguments of polymorphic functions (or in logic terms, to translate existential quantifiers to universal quantifiers at dual position), leading to programs in the same style as handwritten ones following the ad-hoc approach of Chapter 10. The type-class-based solution, like the ad-hoc approach, is not value-independent, because all indexed values need to be declared together in the type class. Also, because each type can only

have one instance of a particular type class, it does not seem likely to support, *e.g.*, defining various formatting functions for the same types of arguments.

It is interesting to note that type classes and value-independent types (or type encodings) form two dimensions of extensibility.

- A type class fixes the set of indexed values, but the types in the type classes can be easily extended by introducing new instances.
- A value-independent type fixes the family of types, but new values indexed by the family can be defined without changing the type declarations.

It would be nice to allow both kinds of extensibility at the same time. But this seems to be impossible—consider the problem of defining a function when possible new types of arguments the function need to handle are not known yet. A linear number of function and type definitions cannot result in a quadratic number of independent variations.

12.4 Conclusion

The approaches above (described in Section 12.1 through Section 12.3) give satisfactory solutions to the problem of type-indexed values. However, since ML-like languages dominate large-scale program development in the functional-programming community, our approach is often immediately usable in common programming practice.

Chapter 13

Concluding remarks for Part II

We have presented a notion of type-indexed values that formalize functions having type arguments. We have formulated type-encoding-based implementations of type-indexed values in terms of type interpretations. According to this formulation, we presented three approaches that enable type-safe programming of type-indexed values in ML or similar languages.

- The first approach directly uses the specific values of a given type-indexed family of values as the type interpretation. It gives value-dependent type encodings, not sharable by different families indexed by the same family of types. However, its efficiency makes it a suitable choice both for applications where all type-indexed values using the same family of types are known in advance, and for the target form of a translation from a source language with explicit support for type-indexed values.
- The second approach is value-independent, abstracting the specification of a type-indexed value from the first approach. Apart from its elegant

form, it may be not very practical because it requires first-class and higher-order polymorphism. But with some efforts, such advanced forms of polymorphism are embeddable in some dialects of ML, such as SML/NJ and Moscow ML.

- The third approach applies the first approach to tune a usual tagged-value-based, type-unsafe approach to give a type-safe and yet syntactically convenient approach, by interpreting types as embedding/projection functions. Though it is less efficient than the first approach due to all the tagging/untagging operations, it allows different type-indexed values to be combined without going beyond the Hindley-Milner type system.

On one hand, we showed in Part II that with appropriate type encodings, type-indexed values can be programmed in ML-like languages; on the other hand, our investigation also feedbacks to the design of new features of type systems. For example, implicit first-class and higher-order polymorphism seem to be useful in applications such as type encodings. The question of what is an expressive enough and yet convenient type system will only be answered by various practical applications.

Concerning programming methodologies, we should note the similarity between type-directed partial evaluation and our third approach in externalizing internal tags. Requiring only a single external tag not only alleviates the burden of manually annotating the program or data with internal tags, but also increases the consistency of these tags. We would like to generalize this idea to other applications.

Part III

The second Futamura projection for type-directed partial evaluation

Chapter 14

Introduction to Part III

14.1 Background

14.1.1 General notions of partial evaluation

First, let us recall the general notions of partial evaluation.

Given a general program $\mathsf{p} : \sigma_S \times \sigma_D \to \sigma_R$ and a fixed *static* input $\mathsf{s} : \sigma_S$, partial evaluation (a.k.a. program specialization) yields a specialized program $\mathsf{p_s} : \sigma_D \to \sigma_R$. When this specialized program $\mathsf{p_s}$ is applied to an arbitrary *dynamic* input $\mathsf{d} : \sigma_D$, it produces the same result as the original program applied to the complete input (s, d), i.e., $[\![\mathsf{p_s d}]\!] = [\![\mathsf{p}(\mathsf{s}, \mathsf{d})]\!]$ (Here, $[\![\cdot]\!]$ maps a piece of program text to its denotation. In this part of the dissertation, meta-variables in *slanted serif* font, such as p, s, and d stand for program terms. Variables in *italic* font, such as x and y, are normal variables in the subject program). Often, some computation in program p can be carried out independently of the dynamic input d, and hence the specialized program $\mathsf{p_s}$ is more efficient than

the general program p. In general, specialization is carried out by performing the computation in the source program p that depends only on the static input s, and generating program code for the remaining computation (called residual code). A partial evaluator PE is a program that performs partial evaluation automatically, i.e., if PE terminates on p and s then

$$[\![PE(p, s)]\!] = p_s$$

Often extra annotations are attached to p and s so as to pass additional information to the partial evaluator.

A program p' is a generating extension of the program p, if running p' on s yields a specialization of p with respect to the static input s (under the assumption that p' terminates on s). Because the program $\lambda s.PE(p, s)$ computes a specialized program p_s for any input s, it is a trivial *generating extension* of program p. To produce a more efficient generating extension, we can specialize PE with respect to p, viewing PE itself as a program and p as part of its input. In the case when the partial evaluator PE itself is written in its input language, i.e., if PE is *self-applicable*, this specialization can be achieved by PE itself. That is, we can generate an efficient generating extension of p as

$$[\![PE(PE, p)]\!].$$

14.1.2 Self-application

The above formulation was first given in 1971 by Futamura [35] in the context of compiler generation—the generating extension of an interpreter is a compiler—and is called the *second Futamura projection*. Turning it into practice, however,

proved to be much more difficult than what its seeming simplicity suggests; it was not until 1985 that Jones's group implemented Mix [59], the very first effective self-applicable partial evaluator. They identified the reason for previous failures: The decision whether to carry out computation or to generate residual code generally depends on the static input s, which is not available during self-application; so the specialized partial evaluator still bears this overhead of decision-making. They solved the problem by taking the decision *offline*, i.e., the source program p is pre-annotated with binding-time annotations that solely determine the decisions of the partial evaluator. In the simplest form, a binding time is either static, which indicates computation carried out at partial-evaluation time (hence called static computation), or dynamic, which indicates code generation for the specialized program.

Subsequently, a number of self-applicable partial evaluators have been implemented, e.g., Similix [9], but most of them are for untyped languages. For typed languages, the so-called *type specialization* problem arises [54]: Generating extensions produced using self application often retain a universal data type and the associated tagging/untagging operations as a source of overhead. The universal data type is necessary for representing static values in the partial evaluator, just as it is necessary for representing values in a standard evaluator. This is unsurprising, because a partial evaluator acts as a standard evaluator when the complete input is static.

Partly because of this, in the 1990's, the practice shifted towards hand-written generating-extension generators [7, 48]; this is also known as the *cogen approach*. Conceptually, a generating-extension generator is a staged partial evaluator, just

as a compiler is a staged interpreter. Ideally, producing a generating extension through self-application of the partial evaluation saves the extra effort in staging a partial evaluator, since it reuses both the technique and the correctness argument of the partial evaluator. In practice, however, it is often hard to make a partial evaluator (or a partial-evaluation technique, as in the case of Part III) self-applicable in the first place. In terms of the correctness argument, if the changes to the partial evaluator in making it self-applicable are minor and are easily proved to be meaning-preserving, then the correctness of a generating extension produced by self-application still follows immediately from that of the partial evaluator.

As we shall see in this work, the problem caused by using a universal data type can be avoided to a large extent, if we can avoid introducing an implicit interpreter in the first place. The second Futamura projection thus still remains a viable alternative to the hand-written approach, as well as a source of interesting problems and a benchmark for partial evaluators.

14.1.3 Type-directed partial evaluation

As we mentioned in the introductionof this dissertation, type-directed partial evaluation (TDPE) performs program specialization by using an efficient reduction-free normalization algorithm, namely the one formalized as examples in the earlier parts of this dissertation. It is different from a traditional, syntax-directed offline partial evaluator [57] in several respects:

Binding-Time Annotation In a traditional partial evaluation setting, all sub-

expressions require binding-time annotations. It is unrealistic for the user to annotate the program fully by hand. Fortunately, these annotations are usually computed by an automatic binding-time analyzer, while the user only needs to provide binding-time annotations on input arguments. On the other hand, since the user does not have direct control over the binding-time annotations, he often needs to know how the binding-time analyzer works and to tune the program in order to ensure termination and a good specialization result.

In contrast, TDPE eliminates the need to annotate expression forms that correspond to function, product and sum type constructions. One only needs to give a binding-time classification for the base types appearing in the types of constants. Consequently, it is possible, and often practical, to annotate the program by hand.

Role of Types The extraction function is parameterized over the type of the term to be normalized, which makes TDPE "type-directed".

Efficiency A traditional partial evaluator works by symbolic computation on the source programs; it contains an evaluator to perform the static evaluation and code generation. TDPE reuses the underlying evaluator (interpreter or compiler) to perform these operations; when run on a highly optimized evaluation mechanism, TDPE acquires the efficiency for free—a feature shared with the cogen approach.

Flexibility Traditional partial evaluators need to handle all constructs used in a subject program, evaluating the static constructs and generating code for

the dynamic ones. In contrast, TDPE uses the underlying evaluator for the static part. Therefore, all language constructs can be used in the static part of a subject program. However, we shall see that this flexibility is lost when self-applying TDPE.

These differences have contributed to the successful application of TDPE in various contexts, e.g., to perform semantics-based compilation [24]. An introductory account, as well as a survey of various treatments concerning Normalization by Evaluation (NbE), can be found in Danvy's lecture notes [15].

14.2 Our work

14.2.1 The problem

A natural question is whether one can perform self-application, in particular the second Futamura projection, in the setting of TDPE. It is difficult to see how this can be achieved, due to the drastic differences between TDPE and traditional partial evaluation.

- TDPE extracts the normal form of a term according to a type that can be assigned to the term. This type is supplied in some form of encoding as an argument to TDPE. We can use self-application to specialize TDPE with respect to a particular type; the result helps one to visualize a particular instance of TDPE. This form of self-application was carried out by Danvy in his original article on TDPE [12]. However, it does not correspond to the second Futamura projection, because no further specialization with respect

to a particular subject program is carried out.

- The aforementioned form of self-application [12] was carried out in the dynamically typed language Scheme. It is not immediately clear whether self-application can be achieved in a language with Hindley-Milner type system, such as ML [71]: Whereas TDPE can be implemented in Scheme as a function that takes a type encoding as its first argument, this strategy is impossible in ML, because such a function would require a dependent type. Indeed, the ML implementation of TDPE uses the technique of type encodings developed in Part II: For every type, a particular TDPE program is constructed. As a consequence, the TDPE algorithm to be specialized is not fixed.

- Following the second Futamura projection literally, one should specialize the source program of the partial evaluator. In TDPE, the static computations are carried out directly by the underlying evaluator, which thus becomes an integral part of the TDPE algorithm. The source code of this underlying evaluator might be written in an arbitrary language or even be unavailable. In this case, writing this evaluator from scratch by hand is an extensive task. It further defeats the main point of using TDPE: to reuse the underlying evaluator and to avoid unnecessary interpretive overhead.

TDPE also poses some technical problems for self-application. For example, TDPE treats monomorphically typed programs, but the standard call-by-value TDPE algorithm uses the polymorphically typed control operators shift and reset to perform let-insertion in a polymorphically typed evaluation context.

14.2.2 Our contribution

This part of the thesis addresses all the above issues, and shows how to effectively carry out self-application for TDPE in a language with Hindley-Milner type system. To generate efficient generating extensions, such as compilers, we reformulate the second Futamura projection in a way that is suitable for TDPE.

More technically, for the typed setting, we show how to use TDPE on the combinators that constitute the TDPE algorithm, and consequently on the type-indexed TDPE itself, and how to slightly rewrite the TDPE algorithm, so that we only use the control operators at the unit and boolean types. As a full-fledged example, we derive a compiler for the Tiny language.

Since TDPE is both the tool and the subject program involved in self-application, we bring together the formal results from Part I and the implementation technique from Part II in Chapter 15, and abstract out a setting in which the later development of self application can be clearly executed. Next, Chapter 16 provides an abstract account of our approach to self-application for TDPE, and Chapter 17 details the development in the context of ML. In particular, Chapter 17.6 describes the derivation of the Tiny compiler, and Chapter 17.7 gives some benchmarks for our experiments. Chapter 18 concludes this part. The appendix gives further technical details in the generation of a Tiny compiler (Appendix E). The complete source code of the development presented in this article is available online [39].

Chapter 15

TDPE revisited

This chapter provides both the theoretical and practical setup for the development in the following chapters. On the theoretical side, we abstract the formal results in Part I in a setting that generalizes call-by-name and call-by-value, and accommodates self-application. On the implementation side, we lay out a functor-based implementation of TDPE, which provides a convenient means to switch between different instantiations, and encapsulates the ad-hoc encoding for the type-indexed extraction functions as combinators.

15.1 A general account of TDPE

In Part I, we observed that TDPE for call-by-name and TDPE for call-by-value have much in common. In this section we present an abstract setting of which each version of TDPE is an instance.

15.1.1 Languages

The abstract setting involves four languages:

- *The object language* L, in which the subject program for specialization, and the generated residual program, are written. The language is parameterized over a signature of base types and constants. Observational equivalence of programs can be established using an equational theory, which is sound with respect to all signatures, consisting of base types and constants, and their interpretations. In our examples, L is nPCF for call-by-name and vPCF for call-by-value.

- *The two-level language* L^2, in which binding-time annotated programs are written. The dynamic part of the language corresponds to the object language L. Furthermore, a standard instantiation maps L^2-phrases to L-phrases. This standard instantiation is called the *annotation erasure*, written as $|\cdot|$. It provides the reference program for correctness of code generation.

- The *TDPE-style two-level language* $\mathsf{L}^{\mathsf{tdpe}}$, in which programs are annotated according to the TDPE type system, where only constants carry the annotations in a term. For $\mathsf{L}^{\mathsf{tdpe}}$, there is also a standard, evaluating instantiation (also called erasure), which maps $\mathsf{L}^{\mathsf{tdpe}}$-phrases to L-phrases for the purpose of specifying correctness. The implementation of $\mathsf{L}^{\mathsf{tdpe}}$ is reduced to that of L^2, through a residualization instantiation $\{\!|\cdot|\!\}_{\mathsf{ri}}$ from $\mathsf{L}^{\mathsf{tdpe}}$-phrases to L^2-phrases.

- *The implementation language* $\mathsf{L}^{\mathsf{imp}}$, an instance of L with a base type exp for term representation. A native embedding $\{\!|\cdot|\!\}_\epsilon$ of L^2 into $\mathsf{L}^{\mathsf{imp}}$ provides an adequate implementation of L^2 in $\mathsf{L}^{\mathsf{imp}}$.

Table 15.1 summarizes the notations and judgment forms that are relevant to our following development, and identifies the corresponding elements for call-by-name and call-by-value. We omit the common ones such as term-in-context.

In the following, for the conciseness of the paper, we will not distinguish L and $\mathsf{L}^{\mathsf{imp}}$, and write L for both of them.

The evaluation function for L, $\mathit{Eval}(\cdot)$, is a computable partial function whose domain consists of all ground-typed terms. That the equational theory of L is sound with respect to observational equivalence implies that provably equal terms can substitute each other without changing program behavior.

Theorem 15.1. *If* $\mathsf{L} \vdash \Delta \rhd E_1 = E_2 : \sigma$*, then for all* σ*-typed contexts* $C[\,]$ *of ground return type,* $\mathit{Eval}(C[E_1]) = \mathit{Eval}(C[E_2])$.

15.1.2 Embedding results

Two key formal results hold in each version of the abstract setting: (1) the TDPE algorithm, which performs normalization for $\mathsf{L}^{\mathsf{tdpe}}$-terms, formulated in L^2, is semantically and syntactically correct; and (2) evaluation of L^2-programs is faithfully simulated in L, through the embedding translation. These results take the form of the following theorems.

Theorem 15.2 (Correctness of TDPE in L^2). *If* $\mathsf{L}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^\emptyset$ *and* $\mathsf{L}^2 \vdash \mathsf{NF}(E) \searrow \mathcal{O}$*, then*

Notation	Comment	Call-by-name	Call-by-value
L	one-level language	nPCF	vPCF
$\mathsf{L} \vdash \Delta \rhd E_1 = E_2 : \sigma$	equation-in-context		
$\mathsf{L} \vdash \Delta \rhd^{nf} E : \sigma$	normal-form-in-context	$\Delta \rhd^{nf} E : \sigma$	$\Delta \rhd^{nc} E : \sigma$
L^2	two-level language	nPCF^2	vPCF^2
$\bigcirc\sigma$	code type	$\bigcirc\sigma$	$\textcircled{e}\sigma$
$\mathsf{L}^2 \vdash E \searrow \mathcal{O}$ (where $\rhd E : \bigcirc\sigma$)	evaluation of a complete program with code type	$E \Downarrow \mathcal{O}$	$E \searrow \mathcal{O}$
$\mathsf{L}^{\mathsf{tdpe}}$	TDPE language	$\mathsf{nPCF}^{\mathsf{tdpe}}$	$\mathsf{vPCF}^{\mathsf{tdpe}}$
$\mathsf{L}^{\mathsf{imp}}$	implementation language	nPCF^{Λ}	$\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$
$Eval(\cdot)$	evaluation function of complete programs	from $[\![\cdot]\!]$	from $\Downarrow$
$\lvert\cdot\rvert$	standard instantiation		
$\{\!\{\cdot\}\!\}_{\mathsf{ri}}$	residual instantiation		
$\{\!\{\cdot\}\!\}_{\epsilon}$	native embedding	$\{\!\{\cdot\}\!\}_{\mathsf{n}\epsilon}$	$\{\!\{\cdot\}\!\}_{\mathsf{v}\epsilon}$
$\mathcal{D}(\cdot)$	decoding of generated code	$\mathcal{D}(\cdot)$	$\{\!\{\cdot\}\!\}^{-1}_{\langle\rangle\epsilon}$
$\mathsf{NF}(E) \triangleq \downarrow^{\sigma}\{\!\{E\}\!\}_{\mathsf{ri}}$	normalization algorithm		

Table 15.1: The abstract setting for TDPE, and its two instances

- $\mathsf{L} \vdash \ \rhd \lvert\mathcal{O}\rvert = \lvert E\rvert : \sigma$ *(semantic correctness), and*
- $\mathsf{L} \vdash \ \rhd^{nf} \lvert\mathcal{O}\rvert : \sigma$ *(syntactic correctness).*

Recall that the erasure $\lvert\mathcal{O}\rvert$ of code-type value $\mathcal{O}$ is the generated code.

Proof. For call by name (column 3 of Table 15.1), we combine Theorem 4.12

and Corollary 4.17. For call by value (column 4 of Table 15.1), we combine Theorem C.28 and Corollary C.29. □

Simulation of the execution of a code-typed L^2-program E in L is carried out by running $\{\!|E|\!\}_\epsilon$ in a "wrapping context" $W[\,]$. For call-by-name, $W[\,] \equiv [\,](1)$; for call-by-value, $W[\,] \equiv \mathsf{letBind}([\,])$.

Theorem 15.3 (Correctness of embedding). *If* $\mathsf{L}^2 \vdash \triangleright E : \bigcirc\sigma$, *then for all code-typed value* $\mathcal{O}$, *the following statements are equivalent.*

(a) $\mathsf{L}^2 \vdash E \searrow \mathcal{O}$;

(b) $\mathcal{D}(Eval(W[\{\!|E|\!\}_\epsilon])) \sim_\alpha |\mathcal{O}|$.

Combining the two preceding theorems yields a native implementation of TDPE algorithm in L. The normal form of a fully-dynamic $\mathsf{L}^{\mathsf{tdpe}}$-term $E : \sigma^\emptyset$, denoted as $NF(E)$, is extracted as

$$\mathcal{D}(Eval(W[\{\!|\mathsf{NF}(E)|\!\}_\epsilon])) \equiv \mathcal{D}(Eval(W[\{\!|\downarrow^\sigma \{\!|E|\!\}_{\mathsf{ri}}|\!\}_\epsilon])).$$

Because both $\{\!|\cdot|\!\}_\epsilon$ and $\{\!|\cdot|\!\}_{\mathsf{ri}}$ are instantiations, they distribute over the sub-terms, and their composition is also an instantiation. Writing $\downarrow^\sigma$ for $\{\!|\downarrow^\sigma|\!\}_\epsilon$, $\uparrow_\sigma$ for $\{\!|\uparrow_\sigma|\!\}_\epsilon$, $\overline{E}$ for $\{\!|\{\!|E|\!\}_{\mathsf{ri}}|\!\}_\epsilon$, $\mathcal{D}[\![E]\!]$ for $\mathcal{D}(Eval(E))$, and omitting the rather simple context $W[\,]$ for brevity, we have $NF(E) = \mathcal{D}(Eval(\{\!|\downarrow^\sigma|\!\}_\epsilon \{\!|\{\!|E|\!\}_{\mathsf{ri}}|\!\}_\epsilon)) = \mathcal{D}[\![\downarrow^\sigma \overline{E}]\!]$.

The instantiation $\overline{E}$, unlike $\{\!|E|\!\}_{\mathsf{ri}}$, is into the one-level language L, but we shall call it the residualizing instantiation of E in the subsequent text. Working out the details of the composition, we can define $\overline{\cdot}$ as follows.

Definition 15.4 (Residualizing Instantiation). *The* residualizing instantiation *of a* $\mathsf{L}^{\mathsf{tdpe}}$*-term* $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright \mathsf{e} : \tau$ *in* L *is* $\mathsf{L} \vdash \triangleright \overline{\mathsf{e}} : \overline{\tau}$, *given by* $\overline{\mathsf{e}} = \mathsf{e}\{\Phi_\sqcup\}$

and $\ulcorner\tau\urcorner = \tau\{\Phi_{\sqcap}\}$*, where instantiation* $\Phi_{\sqcap}$ *is a substitution of* $\mathsf{L}^{\mathsf{tdpe}}$*-constructs into* L*-phrases: for base types* b*,* $\Phi_{\sqcap}(\mathsf{b}) = \mathsf{b}$*,* $\Phi_{\sqcap}(\mathsf{b}^{\mathfrak{d}}) = \mathsf{exp}$*, for constants* c*,* $\Phi_{\sqcap}(c) = c$*,* $\Phi_{\sqcap}(c^{\mathfrak{d}} : \sigma^{\mathfrak{d}}) = \uparrow_{\sigma} \mathsf{CST}(\wr d \wr)$*, and for lifting functions over a base type* b*,* $\Phi_{\sqcap}(\$_{\mathsf{b}}) = \mathsf{LIT}_{\mathsf{b}}$.

For the conciseness of presentation, we have made a few non-essential changes in the above definition, such as unfolding a few terms, and treating the lifting constructs in $\mathsf{L}^{\mathsf{tdpe}}$ as constants. Also, for clarity, we start to use the following meta-variable convention: meta-variable t range over one-level terms, e over two-level terms, σ over one-level types, and τ over two-level types.

To connect with the informal introduction to TDPE in Section 2.4.2, let us take a closer look at the residualizing instantiation. In words, the residualizing instantiation $\ulcorner\tau\urcorner$ of a $\mathsf{L}^{\mathsf{tdpe}}$-type τ substitutes the type exp for all occurrences of dynamic base types in τ. When type τ is fully dynamic, the type $\ulcorner\tau\urcorner$ is constructed solely from type exp, and thus represents a code value or a code manipulation function (see Section 2.4.2). The residualizing instantiation $\ulcorner e \urcorner$ of a term e substitutes all the occurrences of dynamic constants and lifting functions with the corresponding code-generation versions (cf. Example 2.6 (page 23), where $\mathtt{height}_{\mathtt{ann}}$ is $\lambda(a : \mathtt{real}).\lambda(z : \mathtt{real}^{\mathfrak{d}}).\mathtt{mult}^{\mathfrak{d}}\ (\$_{\mathtt{real}}(\mathtt{sin}\ a))\ z)$.

To be consistent, we also present the evaluating instantiation in substitutional form.

Definition 15.5 (Evaluating Instantiation). *The* evaluating instantiation *of a* $\mathsf{L}^{\mathsf{tdpe}}$*-term* $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright e : \tau$ *in* L *is* $\mathsf{L} \vdash \triangleright|e| : |\tau|$*, given by* $|e| = e\{\Phi_{||}\}$ *and* $|\tau| = \tau\{\Phi_{||}\}$*, where instantiation* $\Phi_{||}$ *is a substitution of* $\mathsf{L}^{\mathsf{tdpe}}$*-constructs*

(constants and base types) into L*-phrases (terms and types):* $\Phi_{||}(\mathsf{b}) = \Phi_{||}(\mathsf{b}^{\mathsf{0}}) = \mathsf{b}$, $\Phi_{||}(c) = \Phi_{||}(c^{\mathsf{0}}) = c$, $\Phi_{||}(\$_{\mathsf{b}}) = \lambda x.x$.

Putting everything together, we have the following correctness theorem for the direct formulation of the TDPE algorithm in L. For conciseness, we omit the fixed outer context $W[\,]$ in the formulation.

Theorem 15.6 (Correctness of TDPE in L). *The function NF defined in Equation (15.1) in Figure 15.1 is a static normalization function for* $\mathsf{L}^{\mathsf{tdpe}}$ *in* L. *That is, for* $\mathsf{L}^{\mathsf{tdpe}} \vdash \rhd \mathsf{e} : \sigma^{\mathsf{0}}$, *if* $\mathsf{t} = NF(\mathsf{e})$, *then*
(a) $\mathsf{L} \vdash \rhd |\mathsf{e}| = \mathsf{t} : \sigma$; *and*
(b) $\mathsf{L} \vdash \rhd^{nf}\ \mathsf{t} : \sigma$.

Proof. We combine the previous two theorems. □

Just as self-application reduces the technique of producing an efficient generating extension to the technique of partial evaluation, our results on the correctness of self-application reduce to Theorem 15.6.

15.1.3 Partial evaluation

Given a $\mathsf{L}^{\mathsf{tdpe}}$-term $\mathsf{L}^{\mathsf{tdpe}} \vdash \rhd \mathsf{p} : \tau_S \times \tau_D \to \tau_R$, and its static input $\mathsf{L}^{\mathsf{tdpe}} \vdash \rhd \mathsf{s} : \tau_S$, where both type τ_D and type τ_R are fully dynamic, specialization can be achieved by applying NbE (Equation (15.1) in Figure 15.1) to statically normalize the trivial specialization $\lambda x.\mathsf{p}(\mathsf{s}, x)$:

$$\begin{aligned} NF(\lambda x.\mathsf{p}(\mathsf{s}, x)) &= \mathcal{D}[\![\downarrow^{\tau_D \to \tau_R} \overline{\lambda x.\mathsf{p}(\mathsf{s}, x)}\,]\!] \\ &= \mathcal{D}[\![\downarrow^{\tau_D \to \tau_R} \lambda x.\overline{\mathsf{p}}(\overline{\mathsf{s}}, x)]\!] \end{aligned} \tag{15.2}$$

Normalization by Evaluation

For term $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright e : \sigma^{\mathfrak{d}}$, we use

$$NF(e) = \mathcal{D}[\![\downarrow^{\sigma} \ulcorner e \urcorner]\!] \tag{15.1}$$

to compute its static normal form, where

1. Term $\mathsf{L} \vdash \triangleright\ulcorner e \urcorner : \ulcorner \sigma^{\mathfrak{d}} \urcorner$ is the residualizing instantiation of term e, and
2. Term $\mathsf{L} \vdash \triangleright\downarrow^{\sigma} : \ulcorner \sigma^{\mathfrak{d}} \urcorner \to \mathsf{exp}$ is the one-level reification function for type τ.

Binding-time annotation The task is, given $\mathsf{L} \vdash \triangleright t : \sigma$ and binding-time constraints in the form of a two-level type τ whose erasure is σ, to find $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright t_{\mathsf{ann}} : \tau$ that satisfies the constraints and makes the following equation provable:

$$\mathsf{L} \vdash \triangleright |t_{\mathsf{ann}}| = t : \sigma$$

Figure 15.1: A formal recipe for TDPE

In the practice of partial evaluation, one usually is not given two-level terms to start with. Instead, we want to specialize ordinary programs. This can be reduced to the specialization of two-level terms through a binding-time annotation step. For TDPE, the task of binding-time annotating a L-term t with respect to some knowledge about the binding-time information of the input is, in general, to find a two-level term t_{ann} such that (1) the evaluating instantiation $|t_{\mathsf{ann}}|$ agrees with t, i.e., they are equal in the theory of L, and (2) term t_{ann} is compatible with the input's binding-time information in the following sense: Forming the application of t_{ann} to the static input results in a term of fully dy-

namic type. Consequently, the resulting term can be normalized with the static normalization function NF.

Consider again the standard form of partial evaluation. We are given a L-term $\mathsf{L} \vdash \triangleright p : \sigma_S \times \sigma_D \to \sigma_R$ and the binding-time information of its static input s of type σ_S, but not the static input s itself. The binding-time information can be specified as a $\mathsf{L}^{\mathsf{tdpe}}$-type τ_S such that $|\tau_S| = \sigma_S$; for the more familiar first-order case, type σ_S is some static base type b, and type τ_S is simply b. We need to find a two-level term $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright p_{\mathsf{ann}} : \tau_S \times \tau_D \to \tau_R$, such that (1) types τ_D and τ_R are the fully dynamic versions of types σ_D and σ_R: $\tau_D = \sigma_D^{\mathfrak{d}}$ and $\tau_R = \sigma_R^{\mathfrak{d}}$, and (2) $\mathsf{L} \vdash \ \triangleright \ |p_{\mathsf{ann}}| = p : \sigma_S \times \sigma_D \to \sigma_R$. In most cases, we can require strict equality: $|p_{\mathsf{ann}}| \equiv p$; that is, without using any "binding-time improvement".

For a given static input $s : \sigma_S$, we want to normalize term $t \triangleq \lambda x.p(s, x)$. Given a properly annotated $s_{\mathsf{ann}} : \tau_S$ ("properly" in the sense that $|s_{\mathsf{ann}}| = s$), we can form the two-level term $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright t_{\mathsf{ann}} \triangleq \lambda x.p_{\mathsf{ann}}(s_{\mathsf{ann}}, x) : \tau_D \to \tau_R$. By congruence of the equational theory, $\mathsf{L}^{\mathsf{tdpe}} \vdash \ \triangleright \ t_{\mathsf{ann}} = t : \sigma_D \to \sigma_R$. If the term $t' = NF(t_{\mathsf{ann}})$ is the result of the NbE algorithm, we have, by Theorem 15.6 (page 159), that the following equation is provable in L.

$$t' = |t_{\mathsf{ann}}| = t$$

This verifies the correctness of the specialization.

15.2 TDPE in ML

15.2.1 The setting

To be concrete, we will present the implementation of our development in ML. In the notation of Section 15.1, the one-level language L is ML, with a base type exp for encoding term representations, the constructors associated with exp, constants for name generations, etc. All of these can be introduced into ML as user-defined data types and functions; in practice, we do not distinguish between L and ML. The associated two-level languages L^2 (which we do not need to explicitly use anymore) and $\mathsf{L}^{\mathsf{tdpe}}$ are determined through the signature of L.

15.2.2 Implementation

We use the type-encoding technique developed in Part II to implement the type-indexed functions such as reification and reflection. To recall briefly, an encoding combinator $\langle\!|c|\!\rangle$ is defined for every type constructor c (e.g., $\bullet$ and $\rightarrow$ in the simple setting of Pure NbE). For the indexed family of reification-reflection functions, each combinator takes a *pair* of reification and reflection functions for every argument τ_i to the (n-ary) type constructor c, and computes the reification-reflection pair for the constructed type $c(\tau_1, \ldots, \tau_n)$. Reification and reflection functions for a certain type τ can then be created by combining the combinators according to the structure of τ and projecting out either the reification or the reflection function.

We first illustrate the structure of the implementation using the minimalistic setting of Pure NbE (Section 2.4.2). As Figure 15.2 shows, we specify

```
signature NBE =                                    (* normalization by evaluation *)
  sig
    type Exp
    type 'a rr                                      (* (↓^τ, ↑_τ) : τ rr *)

    val a' : Exp rr                                          (* τ = • *)
    val --> : 'a rr * 'b rr -> ('a -> 'b) rr        (* τ = τ1 → τ2 *)
      ⋮
    val reify: 'a rr -> 'a -> Exp                            (* ↓^τ *)
    val reflect: 'a rr -> Exp -> 'a                          (* ↑_τ *)
  end

signature EXP =                                    (* term representation *)
  sig
    type Exp
    type Var

    val VAR: Var -> Exp
    val LAM: Var * Exp -> Exp
    val APP: Exp * Exp -> Exp
     ⋮
  end

signature GENSYM =                                 (* name generation *)
  sig
    type Var
    val new: unit -> Var                           (* make a new name *)
    val init: unit -> unit                         (* reset name counter *)
  end;
```

Figure 15.2: NbE in ML, signatures

these combinators in a signature called `NBE`. Their implementation as the functor `makePureNbE`—parameterized over two structures of respective signatures `EXP` (term representation) and `GENSYM` (name generation for variables)—is given in Figure 15.3. The implementation of Pure NbE is a direct transcription from the formulation.

Example 15.7. *We implement an* `NBE`*-structure* `PureNbE` *by applying the functor*

```
functor makePureNbE(structure G: GENSYM
                    structure E: EXP
                    sharing type E.Var = G.Var): NBE =
  struct
    type Exp = E.Exp
    datatype 'a rr = RR of ('a -> Exp) * (Exp -> 'a)
                                                    (* (↓^τ, ↑_τ) : τ rr *)
    infixr 5 -->
    val a' = RR (fn e => e, fn e => e)                          (* τ = • *)
    fun (RR (reif1, refl1)) --> (RR(reif2, refl2))          (* τ = τ1 → τ2 *)
      = RR (fn f =>
              let val x = G.new ()
                in E.LAM (x, reif2 (f (refl1 (E.VAR x))))
              end,
            fn e =>
              fn v => refl2 (E.APP (e, reif1 v)))

    fun reify (RR (reif, refl)) v                               (* ↓^τ *)
      = (G.init (); reif v)
    fun reflect (RR (reif, refl)) e                             (* ↑_τ *)
      = refl e
  end
```

Figure 15.3: Pure NbE in ML, implementation

`makePureNbE` *(Figure 15.3); this provides us with combinators* `-->` *and* `a'` *and functions* `reify` *and* `reflect`. *Normalization of* KK *(see Example 2.2 (page 17) and Example 2.3 (page 19)) is carried out as follows:*

```
local  open PureNbE; infixr 5 -->  in

  val K = (fn x => fn y => x)

  val KK_norm = reify (a' --> a' --> a' --> a') (K K)

end
```

After evaluation, the variable `KK_norm` *is bound to a term representation of the normal form of* KK.

15.2.3 Encoding two-level terms through functors

As mentioned earlier, the input to TDPE is a two-level term in $\mathsf{L}^{\mathsf{tdpe}}$. The ML module system makes it possible to encode a two-level term p in a convenient way: Define p inside a functor `p_pe(structure D: DYNAMIC) = ...` which parameterizes over all dynamic types, dynamic constants and lifting functions. By instantiating `D` with an appropriate structure, one can create either the evaluating instantiation $|p|$ or the residualizing instantiation $\ulcorner p \urcorner$.

Example 15.8. *In Example 2.6 (page 23) we sketched how the function* `height` *can be partially evaluated with respect to its first argument. Figure 15.4 shows how to provide both evaluating and residualizing instantiation in ML using functors. We encode the term* $\texttt{height}_{\mathsf{ann}}$ *as a functor* `height_pe(structure D:DYNAMIC)` *that is parameterized over the dynamic type* `Real`*, the dynamic constant* `mult`*, and the lifting function* `lift_real` *in* $\texttt{height}_{\mathsf{ann}}$.

15.2.4 Extensions

We will use an extended version of TDPE, referred to as *Full TDPE* in Part III. Full TDPE not only treats the function type constructor, but also tuples and sums.

Extending TDPE to tuples is straightforward: reifying a tuple is done by producing the code of a tuple constructor and applying it to the reified components of the tuple; reflection at a tuple type means producing code for a projection on every component, reflecting these code pieces at the corresponding component type and tupling the results.

```
signature DYNAMIC =                    (* Signature of dynamic types and constants *)
  sig
    type Real

    val mult: Real -> Real -> Real
    val lift_real: real -> Real
  end

                                       (* The functor encodes a two-level term *)
functor height_pe(structure D: DYNAMIC) =
 struct
   fun height a z = D.mult (D.lift_real (sin a)) z
 end


structure EDynamic: DYNAMIC =                          (* Defining | · | *)
  struct
    type Real = real
    fun mult x y  =  x * y
    fun lift_real r = r
  end

structure RDynamic: DYNAMIC =                          (* Defining ·‾ *)
  struct
    local
      open EExp PureNbE
      infixr 5 -->
    in
      type Real = Exp
      val mult = reflect (a' --> a' --> a') (VAR "mult")
      fun lift_real r  = LIT_REAL r
    end
  end


structure Eheight = height_pe (structure D = EDynamic);
                                                   (* |height_ann| *)
structure Rheight = height_pe (structure D = RDynamic);
                                                   (* ‾height_ann‾ *)
```

Figure 15.4: Instantiation via functors

```
signature CTRL =                                          (* control operators *)
  sig
    type Exp
    val shift: (('a -> Exp) -> Exp) -> 'a
    val reset: (unit -> Exp) -> Exp
  end;

functor makeFullNbE(structure G: GENSYM
                    structure E: EXP
                    structure C: CTRL
                    sharing ... ): NBE = ...
```

Signatures `GENSYM`, `EXP`, and `NBE` are defined in Figure 15.2 (page 163).

Figure 15.5: Full NbE in ML

Sum types, as well as computational features which so far were handled using state, can together be handled by manipulating the code-generation context in the reflection function. This has been achieved by using the control operators shift and reset [18, 29]. Section 17.5 describes in more detail the treatment of sum types and call-by-value languages in Full TDPE.

Figure 15.5 displays the signature `CTRL` of control operators and the skeleton of a functor `makeFullNbE`, which implements Full TDPE—an implementation can be found in Danvy's lecture notes [15]. The relevance of Full TDPE in this article is that (1) it is the partial evaluator that one would use for specializing realistic programs; and (2) in particular, it still conforms to the setting of Section 15.1, in that the implementation language is the same as the object language and it handles all features used in its own implementation, including side effects and control effects. Hence in principle self-application should be possible.

Chapter 16

Formulating self-application

In this section, we present two forms of self-application for TDPE. One uses self-application to generate more efficient reification and reflection functions for a type τ; following Danvy [12], we refer to this form of self-application as *visualization*. The other adapts the second Futamura projection to the setting of TDPE. We first give an intuitive account of how self-application can be achieved, and then derive a precise formulation of self-application, based on the formal account of TDPE presented in Section 15.1.

16.1 An intuitive account of self-application

We start by presenting the intuition behind the two forms of self application, drawing upon the informal account of TDPE in Section 2.4.2.

16.1.1 Visualization

For a specific type σ, the reification function $\downarrow^\sigma$ contains one β-redex for each recursive call following the type structure. For example, the direct unfolding of $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$, according to its definition (Figure 2.2, page 19), is

$$\lambda f_0.\underline{\lambda} x.(\lambda f_1.\underline{\lambda} y.(\lambda f_2.\underline{\lambda} z.(\lambda e.e)(f_2((\lambda e.e)z)))(f_1((\lambda e.e)y)))(f_0((\lambda e.e)x))$$

rather than the normalized form presented in Example 2.3 (page 19). Normalization of such a function can be achieved by self-applying TDPE so as to specialize the reification function with respect to a particular type. Danvy has carried out this form of self application in the dynamically typed language Scheme [12]; in the following, we reconstruct it in our setting.

Recall from Section 2.4.2 that finding the normal form of a term $t : \sigma$ is achieved by reifying the residualizing instantiation of a binding-time annotated version of t:

$$NF(t) = \mathcal{D}[\![\downarrow^\sigma \ulcorner t_{\mathsf{ann}} \urcorner]\!].$$

It thus suffices to find an appropriate binding-time annotated version of the term $\downarrow^\sigma$. A straightforward analysis of the implementation of NbE (see Figure 15.2 (page 163) and Figure 15.3 (page 164)), shows that all the base types (`Exp`, `Var`, etc.) and constants (`APP`, `Gensym.init`, etc.[1]) are needed in the code-generation phase; hence they all should be classified as dynamic. Therefore, to normalize $\downarrow^\sigma : \overline{\sigma^{\mathfrak{d}}} \to \mathsf{exp}$, we use a trivial binding-time annotation, noted $\langle\ \underline{\ .\ }\ \rangle$, in which

[1] These constants appear, e.g., in the underlined portion of the expanded term $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$.

every constant is marked as dynamic:

$$NF(\underline{\langle \downarrow^{\sigma} \rangle}) = \mathcal{D}[\![\downarrow^{\sigma \to \bullet} \overline{\underline{\langle \downarrow^{\sigma} \rangle}}]\!], \tag{16.1}$$

In order to understand the term $\overline{\underline{\langle \downarrow^{\sigma} \rangle}}$, we analyze the composite effect of the residualizing instantiation and trivial binding-time annotation: for a term e, the term $\overline{\underline{\langle e \rangle}}$ is formed from e by substituting all constants with their code-generation counterparts. We write $\Downarrow^{\sigma}$ for $\overline{\underline{\langle \downarrow^{\sigma} \rangle}}$ and $\Uparrow_{\sigma}$ for $\overline{\underline{\langle \uparrow_{\sigma} \rangle}}$ for notational conciseness.

Term $\downarrow^{\sigma}$ and term $\Downarrow^{\sigma}$ are respectively the evaluating instantiation and residualizing instantiation of the same (two-level) term $\underline{\langle \downarrow^{\sigma} \rangle}$: that is, $|\underline{\langle \downarrow^{\sigma} \rangle}| = \downarrow^{\sigma}$, and $\overline{\underline{\langle \downarrow^{\sigma} \rangle}} = \Downarrow^{\sigma}$; term $\uparrow_{\sigma}$ and term $\Uparrow_{\sigma}$ have an analogous relationship. We will exploit this fact in Section 17.1 to apply the functor-based approach to the reification/reflection combinators themselves, thus providing an implementation of $\Downarrow^{\sigma}$ and $\Uparrow_{\sigma}$ in ML.

16.1.2 Adapted second Futamura projection

As we have argued in the introduction to Part III, in the setting of TDPE, following the second Futamura projection literally is not a reasonable choice for deriving efficient generating extensions. The evaluator for the language in which we use TDPE might not even be written in this language. Furthermore, making an evaluator explicit in the partial evaluator to be specialized introduces an extra layer of interpretation, which defeats the advantages of TDPE. We thus consider instead the general idea behind the second Futamura projection:

Using partial evaluation to perform the static computations in a 'triv-

ial' generating extension (usually) yields a more efficient generating extension.

Following the informal recipe for performing TDPE given in Section 2.4.2, the 'trivial generating extension' $p^\dagger$ of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ is

$$\lambda s.TDPE(p, s) : \sigma_S \to \mathsf{exp} = \lambda s.{\downarrow}^{\sigma_D \to \sigma_R}\ \lambda d.\overline{p_{\mathsf{ann}}}(s, d)$$

Since the trivial generating extension is itself a term, we can normalize it using TDPE: We reify at type $\sigma_S \to \bullet$ the residualizing instantiation of the (suitably binding-time annotated) trivial generating extension. We can use the trivial binding-time annotation, i.e., to reify $\overline{\langle\, \lambda s.TDPE(p, s) \,\rangle}$—in Chapter 16.2 we shall explain in detail why this choice is actually not too conservative. Because $\overline{\langle\, . \,\rangle}$ is a substitution, it distributes over term constructors, and we can move it inside the terms:

$$\overline{\langle\, \lambda s.TDPE(p, s) \,\rangle} = \lambda s.\ {\Downarrow}^{\sigma_D \to \sigma_R}\ (\lambda d.\ \overline{\langle\, \overline{p_{\mathsf{ann}}} \,\rangle}\,(s, d)).$$

For concreteness, the reader might find it helpful to consider the example of the height function (Example 2.6, page 23): $\overline{p_{\mathsf{ann}}}$ corresponds to $\overline{\mathtt{height}_{\mathsf{ann}}}$, so $\overline{\langle\, \overline{p_{\mathsf{ann}}} \,\rangle}$ is formed by substituting all the constants in $\overline{\mathtt{height}_{\mathsf{ann}}}$ with their code-generation versions. Such constants include `sin`, $\mathsf{LIT}_{\mathtt{real}}$, and the code-constructing constants appearing in term $\mathtt{mult}_{\mathsf{r}}$ (Example 2.5 (page 20)).

In practice, however, we do not need to first build the residualizing version by hand and then apply the TDPE formulation. Instead, we show that we can characterize $\overline{\langle\, \overline{e} \,\rangle}$ in terms of the original two-level term e itself, thus enabling a functor-based approach: We write $\overline{\overline{e}}$ for $\overline{\langle\, \overline{e} \,\rangle}$ and call it the *GE-instantiation* of term e, where "GE" stands for *generating extension*. A precise

definition of the GE-instantiation is derived formally in Chapter 16.2 (Definition 16.6 (page 176)). Basically, $\overline{\text{"}e\text{"}}$ instantiates all static constants and lifting functions in e with their code-generation version and all dynamic constants with versions that generate "code-generation" code. In other words, static constants and lifting functions give rise to code that is executed when applying the generating extension, whereas dynamic constants give rise to code that has to appear in the result of applying the generating extension.

All in all, the generating extension $p^\ddagger$ of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ can be calculated as

$$p^\ddagger = \mathcal{D}[\![\downarrow^{\sigma_S \to \bullet} (\lambda s.\, \Downarrow^{\sigma_D \to \sigma_R} (\lambda d.\overline{\text{"}p_{\mathsf{ann}}\text{"}}\, (s, d)))]\!]. \tag{16.2}$$

16.2 A derivation of self-application

In Section 16.1 we gave an intuitive account of how self-application can be achieved for TDPE. Using the formalization of TDPE presented in Section 15.1 we now derive both forms of self-application; correctness thus follows from the correctness of TDPE.

16.2.1 Visualization

We formally derive visualization (Section 16.1.1), using the "recipe" outlined in Figure 15.1 (page 160). First, we need a formal definition of the trivial binding-time annotation $\langle\, \underline{\ .\ } \,\rangle$ in terms of the two-level language:

Definition 16.1 (Trivial Binding-Time Annotation). *The* trivial binding-time annotation *of a* L*-term* $\mathsf{L} \vdash \triangleright t : \sigma$ *is a* $\mathsf{L}^{\mathsf{tdpe}}$*-term* $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright \langle \underline{t} \rangle : \langle \underline{\sigma} \rangle$,

given by $\langle\underline{\ t\ }\rangle = t\{\Phi_{\langle\rangle}\}$ *and* $\langle\underline{\ \sigma\ }\rangle = \sigma\{\Phi_{\langle\rangle}\}$*, where the instantiation* $\Phi_{\langle\rangle}$ *is a substitution of* L*-constructs into* $\mathsf{L}^{\mathsf{tdpe}}$*-phrases:* $\Phi_{\langle\rangle}(\mathsf{b}) = \mathsf{b}^{\mathfrak{d}}$*,* $\Phi_{\langle\rangle}(\ell : \mathsf{b}) = \$_{\mathsf{b}}\ell$ *(*ℓ *is a literal),* $\Phi_{\langle\rangle}(c) = c^{\mathfrak{d}}$ *(*c *is not a literal).*

Lemma 16.2 (Properties of $\langle\underline{\ .\ }\rangle$). *For a* L*-term* $\mathsf{L} \vdash \triangleright t : \sigma$*, the following properties hold:*

1. $|\langle\underline{\ t\ }\rangle| \equiv t$*, making* $\langle\underline{\ t\ }\rangle$ *a binding-time annotation of* t*;*
2. $\langle\underline{\ \sigma\ }\rangle$ *is always a fully dynamic type;*
3. *For a fully dynamic* $\mathsf{L}^{\mathsf{tdpe}}$*-type* $\sigma^{\mathfrak{d}}$*,* $\overline{\langle\underline{\ \overline{\sigma^{\mathfrak{d}}}\ }\rangle} = \overline{\sigma^{\mathfrak{d}}}$*.*

A simple derivation using the properties (2) and (3) in Lemma 16.2, together with the fact that $\langle\underline{\ .\ }\rangle$ and $\overline{\cdot}$ distribute over all type and term constructors, yields the formulation of self-application given in Equation (16.1) on page 170:

$$NF(\langle\underline{\ \downarrow^{\sigma}\ }\rangle) = \mathcal{D}[\![\downarrow^{\sigma\to\bullet}(\Downarrow^{\sigma})]\!].$$

The following corollary follows immediately from Theorem 15.6 (page 159) and property (1) of Lemma 16.2.

Corollary 16.3. *If* $t_\sigma = NF(\langle\underline{\ \downarrow^{\sigma}\ }\rangle)$*, then*
(a) t_σ *is in normal form:* $\mathsf{L} \vdash \triangleright^{nf} t_\sigma : \overline{\sigma^{\mathfrak{d}}} \to \mathsf{exp}$*; and*
(b) t_σ *is semantically equal to* $\downarrow^{\sigma}$*:* $\mathsf{L} \vdash \downarrow^{\sigma} = t_\sigma$*.*

The self-application carried out by Danvy in the setting of Scheme [12] is quite similar; his treatment explicitly λ-abstracts over the constants occurring in $\downarrow^{\tau}$, which, by the TDPE algorithm, would be reflected according to their types. This reflection also appears in our formulation: For any constant $c : \sigma$

appearing in $\downarrow^\tau$, we have $\overline{\langle c \rangle} = \overline{c^\emptyset} = \uparrow_\sigma \mathsf{CST}(\wr d \wr)$. Consequently, our result coincides with Danvy's.

16.2.2 Adapted second Futamura projection

We repeat the development from Section 16.1.2 in a formal way. We begin by rederiving the trivial generating extension, this time from Equation (15.2) on page 159: In order to specialize a two-level term $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright p : \tau_S \times \tau_D \to \tau_R$ with respect to a static input $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright s : \tau_S$ (where $|\tau_S| = \sigma_S$, $\tau_D = \sigma_D^\emptyset$, and $\tau_R = \sigma_S^\emptyset$), we execute the L-program $\mathsf{L} \vdash \triangleright \downarrow^{\sigma_D \to \sigma_R} \lambda d.\overline{p}(\overline{s}, d) : \mathsf{exp}$. By λ-abstracting over the residualizing instantiation $\overline{s}$ of the static input s, we can trivially obtain a generating extension $p^\dagger$, which we will refer to as the trivial generating extension.

$$\mathsf{L} \vdash \triangleright p^\dagger \triangleq \lambda s.\, \downarrow^{\sigma_D \to \sigma_R} (\lambda d.(\overline{p}(s, d))) : \overline{\tau_S} \to \mathsf{exp}.$$

Corollary 16.4 (Trivial Generating Extension). *The term $p^\dagger$ is a generating extension of program p.*

Since the term $p^\dagger$ is itself a L-term, we can follow the recipe in Figure 15.1 (page 160) to specialize it into a more efficient generating extension. We first need to binding-time annotate the term $p^\dagger$. For the subterm $\downarrow^{\sigma_D \to \sigma_R}$, the analysis in Section 16.1.1 shows that we should take the trivial binding-time annotation. For the subterm $\overline{p}$, the following analysis shows that it is not too conservative to take the trivial binding-time annotation as well. Since $\overline{\cdot} = \Phi_\sqcup$ is an instantiation, i.e., a substitution on dynamic constants and lifting functions, every constant c' in $\overline{p}$ must appear as a subterm of the image of a constant or a lifting

function under the substitution $\Phi_{\sqcap}$. If c' appears inside $\Phi_{\sqcap}(c^{\mathsf{S}}) = \uparrow_{\sigma} \mathsf{CST}(\wr d \wr)$ (where c' could be a code-constructor such as `LAM`, `APP` appearing in term $\uparrow_{\sigma}$) , or $\Phi_{\sqcap}(\$_{\mathsf{b}}) = \mathsf{LIT}_{\mathsf{b}}$, then c' is needed in the code-generation phase, and hence it should be classified as dynamic. If c' appears inside $\Phi_{\sqcap}(c) = c$, then $c' = c$ is an original constant, classified as static assuming the input s is given. Such a constant could rarely be classified as static in $p^{\dagger}$, since the input s is not statically available at this stage.

Taking the trivial binding-time annotation of the trivial generating extension $p^{\dagger}$, we then proceed with Equation (15.1) on page 160 to generate a more efficient generating extension.

$$
\begin{aligned}
p^{\ddagger} &= NF(\underline{\langle\ \lambda s.\downarrow^{\sigma_D \to \sigma_R}\ (\lambda d.(\overline{\ulcorner p \urcorner}\ (s,d)))\ \rangle}) \\
&= \mathcal{D}[\![\downarrow^{\sigma_S \to \bullet}\ \overline{\underline{\langle\ \lambda s.\downarrow^{\sigma_D \to \sigma_R}\ (\lambda d.(\overline{\ulcorner p \urcorner}\ (s,d)))\ \rangle}}\ ]\!] \\
&= \mathcal{D}[\![\downarrow^{\sigma_S \to \bullet}\ (\lambda s.\overline{\underline{\langle \downarrow^{\sigma_D \to \sigma_R} \rangle}}(\lambda d.(\overline{\underline{\langle \overline{\ulcorner p \urcorner} \rangle}}\ (s,d))))]\!]
\end{aligned}
$$

Expressing $\overline{\underline{\langle \overline{\ulcorner p \urcorner} \rangle}}$ as $\overline{\ulcorner p \urcorner}$, and $\overline{\underline{\langle \downarrow^{\sigma_D \to \sigma_R} \rangle}}$ as $\Downarrow^{\sigma_D \to \sigma_R}$, we have

$$
p^{\ddagger} = \mathcal{D}[\![\downarrow^{\sigma_S \to \bullet}\ (\lambda s.\Downarrow^{\sigma_D \to \sigma_R}\ (\lambda d.\overline{\ulcorner p \urcorner}\ (s,d)))]\!],
$$

as originally given in Equation (16.2) on page 172.

The generation of $p^{\ddagger}$ always terminates, even though, in general, the normalization function *NF* may diverge. Recall that the trivial binding-time annotation used in the preceding computation of $p^{\ddagger}$ marks all constants, including all fixed-point operators, as dynamic. Divergence can only happen when the two-level program contains static fixed-point operators.

The correctness of the second Futamura projection follows from Corollary 16.4 (page 174) and Theorem 15.6 (page 159).

Corollary 16.5 (Efficient Generating Extension). *Program* $p^{\ddagger}$ *is a generating extension of* p *in normal form:*

1. $p^{\ddagger}$ *is in normal form:* $\mathsf{L} \vdash \triangleright^{nf}\ p^{\ddagger} : \ulcorner \tau_S \urcorner \to \mathsf{exp}$.

2. $p^{\ddagger}$ *is semantically equal to* $p^{\dagger}$*:* $\mathsf{L} \vdash p^{\ddagger} = p^{\dagger}$.

Proof. We combine Theorem 15.6 (page 159) and the property of trivial binding-time analysis. In particular,

$$\mathsf{L} \vdash p^{\ddagger} = |\underline{\langle\, \lambda s.\downarrow^{\sigma_D \to \sigma_R} (\lambda d.(\overline{p}\ (s,d)))\,\rangle}| = p^{\dagger}$$

That the program $p^{\ddagger}$ is a generating extension of p follows from Corollary 16.4 (page 174). □

Now let us examine how the term $\overline{p}$ is formed. Note that $\overline{p} = \overline{\langle\, \overline{p}\,\rangle} = ((p\{\Phi_{\sqcap}\})\{\Phi_{\langle\rangle}\})\{\Phi_{\sqcap}\} = p\{\Phi_{\sqcap} \circ \Phi_{\langle\rangle} \circ \Phi_{\sqcap}\}$; thus $\overline{\cdot}$ corresponds to the composition of three instantiations, $\Phi_{\sqcap\!\sqcap} = \Phi_{\sqcap} \circ \Phi_{\langle\rangle} \circ \Phi_{\sqcap}$, which is also an instantiation. We call $\Phi_{\sqcap\!\sqcap}$ the generating-extension instantiation (GE-instantiation); a simple calculation gives the following explicit definition.

Definition 16.6 (GE-instantiation). *The* GE-instantiation *of a* $\mathsf{L}^{\mathsf{tdpe}}$*-term* $\mathsf{L}^{\mathsf{tdpe}} \vdash \triangleright e : \tau$ *in* L *is* $\mathsf{L} \vdash \triangleright \overline{e} : \overline{\tau}$ *given by* $\overline{e} = e\{\Phi_{\sqcap\!\sqcap}\}$ *and* $\overline{\tau} = \tau\{\Phi_{\sqcap\!\sqcap}\}$*, where*

instantiation Φ_{m} *is a substitution of* $\mathsf{L}^{\mathsf{tdpe}}$*-constructs into* L*-phrases:*

$$\Phi_{\mathsf{m}}(\mathsf{b}) = \Phi_{\mathsf{m}}(\mathsf{b}^{\mathsf{d}}) = \mathsf{exp}$$

$$\Phi_{\mathsf{m}}(c : \sigma) = \overline{\langle\, c : \sigma \,\rangle} = \uparrow_{\sigma} \mathsf{CST}(\wr c \wr)$$

$$\Phi_{\mathsf{m}}(c^{\mathsf{d}} : \tau) = \overline{\uparrow_{\sigma} \mathsf{CST}(\wr c \wr)} = \Uparrow_{\sigma} \overline{\langle\, \mathsf{CST} \,\rangle}(\mathsf{LIT}_{\mathtt{const}} \wr c \wr)$$

$$\Phi_{\mathsf{m}}(\$_b) = \uparrow_{\bullet \to \bullet} \mathsf{CST}(\wr \mathsf{LIT}_b \wr)$$

Note that at some places, we intentionally keep the $\overline{\cdot}$ form unexpanded, since we can just use the functor-based approach to obtain the residualizing instantiation. Indeed, the GE-instantiation boils down to "taking the residualizing instantiation of the residualizing instantiation". In Section 17.3, we show how to extend the instantiation-through-functor approach to cover GE-instantiation as well.

It is instructive to compare the formulation of the second Futamura projection with the formulation of TDPE (Equation 15.2, page 159). The crucial common feature is that the subject program p is only instantiated, i.e., only the constants are substituted in the program; this feature makes them amenable to a functor-based treatment and frees them from an explicit interpreter. For TDPE, however, static constants are instantiated with their standard instantiation, which makes it possible to use built-in constructs (such as case expressions) in the "static parts" of a program. This is not the case for the second Futamura projection, which causes some inconvenience when applying the second Futamura projection, as we shall see in Chapter 17.6.

Chapter 17

Implementation and benchmarks

In this section we treat various issues arising when implementing the abstract formulation of Chapter 16 in ML. We start with the implementation of the key components for self application, namely the functions $\Downarrow$ and $\Uparrow$, and the GE-instantiation. We then turn to two technical issues. First, we show how to specify the input, especially the types, for the self-application. Second, we show how to modify the full TDPE algorithm, which uses polymorphically typed control operators, such that it is amenable to the TDPE algorithm itself, i.e., amenable to self-application.

17.1 Residualizing instantiation of the combinators

In Chapter 16.1 we remarked that the terms $\downarrow^\sigma$ and $\Downarrow^\sigma$ are respectively the evaluating instantiation and the residualizing instantiation of the same two-level term $\underline{\langle \downarrow^\sigma \rangle}$. We can again use the ML module system to conveniently implement both instantiations. Recall that we formulated reification and reflection as type-

indexed functions, and we implemented them not as a monolithic program, but as a group of combinators, one for each type constructor. These combinators can be plugged together following the structure of a type σ to construct a type encoding as a reification-reflection pair $(\downarrow^\sigma, \uparrow_\sigma)$. To binding-time annotate $(\downarrow^\sigma, \uparrow_\sigma)$ as $(\langle\underline{\downarrow^\sigma}\rangle, \langle\underline{\uparrow_\sigma}\rangle)$, it suffices to parameterize all the combinators over the constants they use: As already mentioned, because $\langle\underline{\ .\ }\rangle$ is a substitution, it distributes over all constructs in a term, marking all the types and constants as dynamic. The combinators, when instantiated with either an evaluating or a residualizing instantiation, can be combined according to a type σ to yield either $(\downarrow^\sigma, \uparrow_\sigma)$ or $(\Downarrow^\sigma, \Uparrow_\sigma)$.

We can directly use the functors `makePureNbE` (Figure 15.3, page 164) and `makeFullNbE` (Figure 15.5, page 167) to produce the instantiations, because these functors are parameterized over the primitives used in the NbE module. Hence, rather than hard-wiring code-generation primitives, this factorization reuses the implementation for producing both the evaluating instantiation and the residualizing instantiation. An evaluating instantiation `EFullNbE` of NbE is produced by applying the functor `makeFullNbE` to the standard evaluating structures `EExp`, `EGensym` and `ECtrl` of the signatures `EXP`, `GENSYM` and `CTRL`, respectively (Figure 17.1—we show the implementations of structures `EExp` and `EGensym`; for structure `ECtrl`, we use Filinski's implementation [29]). Residualizing instantiations `RFullNbE` of Full NbE and `RPureNbE` of Pure NbE result from applying the functors `makePureNbE` and `makePureNbE`, respectively, to appropriate residualizing structures `RGensym`, `RExp`, and `RCtrl` (Figure 17.2, page 181).

For example, in the structure `RExp`, the type `Exp` and the type `Var` are both

```
structure EExp                                  (* Evaluating Inst. | · | on EXP *)
= struct
    type Var = string
    datatype Exp =
        VAR of string                                                    (* v *)
      | LAM of string * Exp                                          (* λx.e *)
      | APP of Exp * Exp                                          (* e1@e2 *)
      | PAIR of Exp * Exp                                          (* (e1,e2) *)
      | PFST of Exp                                                  (* fst *)
      | PSND of Exp                                                  (* snd *)
      | LIT_REAL of real                                          (* $real *)
  end

structure EGensym                          (* Evaluating Inst. | · | on GENSYM *)
= struct
    type Var = string

    local val n = ref 0
    in fun new () = (n := !n + 1;                          (* make a new name *)
                     "x" ^ Int.toString (!n))
       fun init () = n := 0                                (* reset name counter *)
    end
  end;

(* Evaluating Instantiation *)
structure EFullNbE = makeFullNbE (structure G = EGensym
                                  structure E = EExp
                                  structure C = ECtrl): NBE
```

Figure 17.1: Evaluating Instantiation of NbE

instantiated with `EExp.Exp` since they are dynamic base types, and all the code-constructing functions are implemented as functions that generate 'code that constructs code'; here, to assist understanding, we have unfolded the definition of reflection (see also Example 2.4, page 20).

With the residualizing instantiation of reification and reflection at our disposal, we now can perform visualization by following Equation (16.1) on page 170.

Example 17.1. *We show the visualization of* $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$ *(Example 2.3, page 19)*

```
structure RExp: EXP = struct
  type Exp = EExp.Exp
  type Var = EExp.Exp

                                                        (* VAR v  =  VAR@v *)
  fun VAR v = EExp.APP (EExp.VAR "VAR", v)

                                                  (* LAM (v,e)  =  LAM@(v,e) *)
  fun LAM (v, e) = EExp.APP (EExp.VAR "LAM",
                             EExp.PAIR (v, e))
                                                  (* APP (s,t)  =  APP@(s,t) *)
  fun APP (s, t) = EExp.APP (EExp.VAR "APP",
                             EExp.PAIR (s, t))
  ⋮
end

⋮

                                                (* Residualizing Instantiations *)
structure RFullNbE = makeFullNbE (structure G = RGensym
                                  structure E = RExp
                                  structure C = RCtrl): NBE

structure RPureNbE = makePureNbE (structure G = RGensym
                                  structure E = RExp): NBE
```

Figure 17.2: Residualizing Instantiation of NbE

for Pure NbE. Following Equation (16.1) on page 170, we have to compute $\downarrow^{(\bullet\to\bullet\to\bullet\to\bullet)\to\bullet}(\Downarrow^{\bullet\to\bullet\to\bullet\to\bullet})$. *This is done in Figure 17.3; it is not difficult to see that the result matches the execution of the term* `reify (a' --> a' --> a' --> a')` *(see Figure 15.3, page 164). Visualization of the reflection function is carried out similarly.*

```
local open EFullNbE
  infixr 5 -->
  val Ereify_aaaa_a
    = reify ((a'-->a'-->a'-->a') --> a')                (* ↓^((•→•→•→•)→•→•) *)
  open RPureNbE
  infixr 5 -->
  val Rreify_aaaa = reify (a'-->a'-->a'-->a')           (* ⇓^(•→•→•→•→•) *)
in val nf = Ereify_aaaa_a (Rreify_aaaa) end
```

The (pretty-printed) result `nf` is:

$\lambda x_1.$ **let** $r_2 = \mathsf{init}()$ $r_3 = \mathsf{new}()$ $r_4 = \mathsf{new}()$ $r_5 = \mathsf{new}()$

in

$\underline{\lambda} r_3.\underline{\lambda} r_4.\underline{\lambda} r_5.x_1\ r_3\ r_4\ r_5$

end

Figure 17.3: Visualizing $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$

17.2 An example: Church numerals

We first demonstrate the second Futamura projection with the example of the addition function for Church numerals. The definitions for the Church numeral 0_{ch}, successor s_{ch}, and the addition function $+_{\mathsf{ch}}$ in Figure 17.4 are all standard; as the types indicate, they are given as the residualizing instantiation. One can see that partially evaluating the addition function $+_{\mathsf{ch}}$ with respect to the Church numeral $n_{\mathsf{ch}} = s^n_{\mathsf{ch}}(0_{\mathsf{ch}})$ should produce a term $\lambda n_2.\lambda f.\lambda x.f^n(n_2 f x)$; by definition, this is also the functionality of a generating extension of function $+_{\mathsf{ch}}$.

The term $+_{\mathsf{ch}}$ contains no dynamic constants, hence $\overline{\text{"}+_{\mathsf{ch}}\text{"}} = \overline{\text{`}+_{\mathsf{ch}}\text{'}} = +_{\mathsf{ch}}$. Following Equation (16.2) on page 172, we can compute an efficient generating extension $+_{\mathsf{ch}}{}^{\ddagger}$, as shown in Figure 17.4.

```
type 'a num = ('a -> 'a) -> ('a -> 'a)                    (* Type num *)
val c0 : EExp.Exp num
  = fn f => fn x =>  x                                    (* 0_ch : num *)
fun cS (n: EExp.Exp num)
  = fn f => fn x => f (n f x)                             (* s_ch : num → num *)
fun cAdd (m: EExp.Exp num, n: EExp.Exp num)
  = fn f => fn x =>
       m f (n f x)                                        (* +_ch : (num × num) → num *)

local open EFullNbE
  infixr 5 -->
  val Ereify_n_exp
  = reify  (((a' --> a') --> (a' --> a')) --> a')
                                                          (* ↓^{num→•} *)
  open RPureNbE
  infixr 5 -->
  val Rreify_n_n
  = reify (((a'-->a') --> (a'-->a')) -->
           ((a'-->a') --> (a'-->a')))                     (* ⇓^{num→num} *)
in val ge_add
  = Ereify_n_exp (fn m => (Rreify_n_n (fn n =>
    cAdd (m, n))))                                        (* +_ch‡ *)
end;
```

The (pretty-printed) result $+_{\mathsf{ch}}{}^{\ddagger}$ is:

$\lambda x_1.$ **let** $r_2 = \mathsf{init}()$ $r_3 = \mathsf{new}()$ $r_4 = \mathsf{new}()$ $r_5 = \mathsf{new}()$ $r_7 = \mathsf{new}()$

in

$\underline{\lambda} r_3.\underline{\lambda} r_4.\underline{\lambda} r_5.(x_1(\lambda x_6.(r_4 \underline{@} x_6)))$

$(((r_3 \underline{@} (\underline{\lambda} r_7.(r_4 \underline{@} r_7))) \underline{@} r_5)))$

end

For example, applying $+_{\mathsf{ch}}{}^{\ddagger}$ to `(cS (cS (c0)))` generates

$$\lambda x_1.\lambda x_2.\lambda x_3.x_2(x_2(x_1(\lambda x_4.x_2\ x_4)x_3)).$$

Figure 17.4: Church numerals

17.3 The GE-instantiation

We generalize the technique of encoding a two-level term p in ML presented at the end of Section 15.2: We code p inside a functor

```
p_ge(structure S:STATIC structure D:DYNAMIC) = ...
```

that parameterizes over both static and dynamic constants. With suitable instantiations of the structures `S` and `D`, one can thus create not only the evaluation instantiation $|p|$ and the residualizing instantiation $\ulcorner p \urcorner$, but also the GE-instantiation $\ulcorner\!\ulcorner p \urcorner\!\urcorner$. The instantiation table displayed in Table 17.1 summarizes how to write the components of the three kinds of instantiation functors for `S` and `D`. The table follows easily from the formal definitions of $|\cdot|$, $\ulcorner\cdot\urcorner$ and $\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$ via $\Phi_{||}$ (Definition 15.5, page 158), $\Phi_{\ulcorner\urcorner}$ (Definition 15.4, page 157) and $\Phi_{\ulcorner\!\ulcorner\urcorner\!\urcorner}$ (Definition 16.6, page 176), respectively.

		$\lvert\cdot\rvert$	$\ulcorner\cdot\urcorner$	$\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$
S	b	b	b	exp
	$c : \sigma$	c	c	$\uparrow_\sigma \mathsf{CST}(\wr c \wr)$
D	b^∂	b	exp	exp
	$c^\partial : \sigma^\partial$	c	$\uparrow_\sigma \mathsf{CST}(\wr c \wr)$	$\Uparrow_\sigma \ulcorner\langle \mathsf{CST} \rangle\urcorner(\mathsf{LIT}_{\mathtt{const}}\wr c \wr)$
	$\$_\mathsf{b}$	$\lambda x.x$	LIT_b	$\uparrow_{\bullet\to\bullet} \mathsf{CST}(\wr \mathsf{LIT}_b \wr)$

Table 17.1: Instantiation table

Note, in particular, that $|\cdot|$ and $\ulcorner\cdot\urcorner$ have the same instantiation for the static signature; hence we can reuse $\Phi_{||}$ for $\Phi_{\ulcorner\urcorner}$.

Example 17.2. *We revisit the function* `height`*, which appeared in Example 2.6*

```
signature STATIC =                                                (* Σ *)
  sig
    type SReal                                                    (* real *)
    val sin: SReal -> SReal                                       (* sin *)
  end

signature DYNAMIC =                                               (* Σ^∂ *)
  sig
    type SReal                                                    (* real *)
    type DReal                                                    (* real^∂ *)
    val mult: DReal -> DReal -> DReal                             (* mult^∂ *)
    val lift_real: SReal -> DReal                                 (* $_real *)
  end

functor height_ge(structure S: STATIC                             (* height_ann *)
                  structure D: DYNAMIC
                   sharing type D.SReal = S.SReal) =
 struct
   fun height a z = D.mult (D.lift_real (S.sin a)) z
 end

structure GEStatic: STATIC =                                      (* Φ_m on Σ *)
  struct
    local open EExp EFullNbE; infixr 5 --> in
      type SReal = Exp
      val sin = reflect (a' --> a') (VAR "sin")
    end
  end

structure GEDynamic: DYNAMIC =                                    (* Φ_m on Σ^∂ *)
  struct
    local open RExp RFullNbE; infixr 5 --> in
      type DReal = Exp
      val mult = reflect (a' --> a' --> a')
                         (VAR (EExp.STR "mult"))
      fun lift_real r = LIT_REAL r
    end
  end
                                                                  (* "height_ann" *)
structure ge_height = height_ge(structure S = GEStatic
                                structure D = GEDynamic)
```

Figure 17.5: Instantiation via functors

(page 23) and Example 15.8 (page 165). In Figure 17.5 we define the functor `height_ge` *along with signatures* `STATIC` *and* `DYNAMIC`*. Structure* `GEStatic` *and structure* `GEDynamic` *provide the GE-instantiation for the signature* $\mathsf{L}^{\mathsf{tdpe}}$*. The instantiation of* `height_ge` *with these structures gives* $\overline{\texttt{height}_{\mathsf{ann}}}$*. Applying the second Futamura projection as given in Equation (16.2) on page 172 yields*

$$
\begin{array}{l}
\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}() \\
\qquad\qquad r_3 = \mathsf{new}() \\
\qquad \textbf{in} \\
\qquad\quad \underline{\lambda} r_3.\text{“}\mathit{mult}\text{”}\,\underline{@}(\mathit{lift}_{\mathsf{real}}(\mathit{sin}\ x_1))\underline{@} r_3 \\
\qquad \textbf{end}
\end{array}
$$

17.4 Type specification for self-application

The technique developed so far is already sufficient to carry out visualization or the second Futamura projection, at least in an effect-free setting. Still, it requires the user to manually instantiate self-application Equation (16.1) on page 170 and Equation (16.2) on page 172, as we have done for all the preceding examples. In particular, as Example 17.1 (page 180) demonstrates, one needs to use two different sets of combinators for essentially the same type ($\bullet \to \bullet \to \bullet \to \bullet$ in this case), one for the residualizing instantiation of NbE, and the other for the evaluating instantiation. It would be preferable to package the abstract formulation of Equation (16.1) on page 170 and Equation (16.2) on page 172 as program modules themselves, instead of leaving them as templates for the user. In this section, we achieve such an “instantiation-independent” encoding, using the functor-based approach of Section 11.2.

Types are part of the input in both forms of self-application. The user of the module should specify a type σ in a way that is independent of the instantiations; it is the task of the self-application module to choose whether and where to use the residualization instantiation $(\Downarrow^\sigma, \Uparrow_\sigma)$ or the evaluation instantiation $(\downarrow^\sigma, \uparrow_\sigma)$. Since different instantiations have different types, the type argument, even in the form of an encoding of the corresponding extraction functions, cannot be abstracted over at the function level. Recall that the type-indexed functions are formed by plugging together combinators. Specifying a type, therefore, amounts to writing down how combinators should be plugged together, leaving the actual definition of the combinators (i.e., an `NBE`-structure) abstract.

To make the above idea more precise, let us consider the example of visualizing the reification functions. The specification of a type σ should consist of not only the type σ itself, but also a functor that maps a `NBE`-structure `NbE` to the appropriate instantiation of the pair $(\langle\underline{\downarrow^\sigma}\rangle, \langle\underline{\uparrow_\sigma}\rangle)$, which is of type σ `NbE`.rr. This suggests that the type specification should have the following dependent type:

$$\sum \sigma : *. \prod \texttt{NbE} : \texttt{NBE}.(\sigma\ \texttt{NbE}.rr),$$

where $\sum$ is the dependent-sum formation, and $\prod$ is the dependent-product formation.

We can then turn this type into a higher-order signature `VIS_INPUT` in Standard ML of New Jersey, and in turn write a higher-order functor `vis_reify` that performs visualization of the reification function (Figure 17.6). The example visualization in Figure 17.3 (page 182) can be now carried out using the type

```
signature VIS_INPUT =                          (* Signature for a type specification *)
  sig
    type 'a vis_type                  (* Type σ, parameterized at the base type *)
    functor inp(NbE: NBE) :                        (* parameterized type coding *)
      sig
        val T_enc: (NbE.Exp vis_type) NbE.rr
      end
  end

functor vis_reify (P: VIS_INPUT) =
  struct
    local
      structure eVIS                                 (* Evaluating instantiation *)
              = P.inp(EFullNbE)
      structure rVIS                                 (* Residualizing instantiation *)
              = P.inp(RPureNbE)
      open EFullNbE
      infixr 5 -->
    in
      val vis = reify (eVIS.T_enc --> a')                        (* ↓^σ→• (⇓^σ) *)
                      (RPureNbE.reify rVIS.T_enc)
    end
  end
```

Figure 17.6: Specifying types as functors

specification given in Figure 17.7.

17.5 Monomorphizing control operators

So far we have shown how to self-apply Pure TDPE. When self-applying Full TDPE, one complication arises: The implementation of Full TDPE uses control operators polymorphically in the definition of reflection, but to determine the residualizing instantiation of a constant, a fixed monomorphic type has to be determined. This section shows how to rewrite the algorithm for full TDPE

```
structure a2a : VIS_INPUT =                                  (* A type specification *)
  struct
    type 'a vis_type = 'a->'a->'a->'a                  (* σ = • → • → • → • *)
    functor inp(NbE: NBE) =                                              (* NbE *)
      struct
        local open NbE  infixr 5 --> in
          val T_enc = a' --> a' --> a' --> a'                    (* σ NbE.rr *)
        end
      end
    end

structure vis_a2a = vis_reify(a2a);                          (* Visualization *)
```

Figure 17.7: Type specification for visualizing $\downarrow^{\bullet\to\bullet}$

such that all control operators occur monomorphically.

17.5.1 Let-insertion via control operators

Full TDPE treats call-by-value languages with computational effects. In this setting, *let-insertion* [9, 44] is a standard partial-evaluation technique to prevent duplicating or discarding computations that have side-effects: All computations that might have effects are bound to a variable and sequenced using the (monadic) let construct. When the TDPE algorithm identifies the need to insert a let-construct, however, it usually is not at a point where a let-construct can be inserted, i.e., a code-generating expression.

Using a technique that originated in the study of continuation-based partial evaluation [67], Danvy [11] solves this problem by using the control operators shift and reset [18]: Intuitively speaking, the operator shift abstracts the current evaluation context up to the closest delimiter reset and passes the abstracted context to its argument, which can then invoke this delimited evaluation context

just like a normal function. Formally, the semantics of shift and reset is expressed in terms of the CPS transformation (Figure 17.8; see Danvy and Filinski [18] and Filinski [29] for more details, and Danvy and Yang [25] for an operational account).

$$[\![\mathsf{shift}\ E]\!]_{\mathsf{CPS}} = \lambda\kappa.[\![E]\!]_{\mathsf{CPS}}(\lambda f.f(\lambda v.\lambda\kappa'.\kappa'(\kappa\ v))(\lambda x.x))$$

$$[\![\mathsf{reset}\langle E\rangle]\!]_{\mathsf{CPS}} = \lambda\kappa.\kappa([\![E]\!]_{\mathsf{CPS}}(\lambda x.x))$$

Term $\langle E\rangle$, "the thunk of E", is shorthand for $\lambda().E$. The use of a thunk here delays the computation of E and avoids the need to implement reset as a macro.

Figure 17.8: The CPS semantics of shift/reset

With the help of these control operators, Danvy's treatment [11] follows the following strategy for let-insertion: (1) use reset to 'mark the boundaries' for code generation, i.e., to surround every expression that has type `Exp` and could potentially be a point where let-bindings need to be inserted;[1] (2) when let-insertion is needed, use shift to 'grab the context up to the marked boundary' and bind it to a variable k (thus k is a code-constructing context); (3) apply k to the intended return value to form the body expression of the let-construct, and then wrap it with the let-construct. The new definitions for the reification and reflection functions as given by Danvy are shown in Figure 17.9; there are two function-type constructors: a function type without effects $\sigma_1 \to \sigma_2$, which does

[1] An effect-typing system can provide a precise characterization of where reset has to be used. Roughly speaking, an operator reset encloses the escaping control effect introduced by an inner shift. See Filinski's work [30] for more details.

not require let-insertion, and a function type with possible latent effects $\sigma_1 \xrightarrow{!} \sigma_2$, which does require let-insertion. We extend the type exp of code representations with a constructor `LET of string * Exp * Exp` and write $\underline{\mathbf{let}}\ x \Leftarrow t_1\ \underline{\mathbf{in}}\ t_2$ for `LET (`x`,`t_1`,`t_2`)`; we implement a new TDPE combinator `-!>` in ML for the new type constructor $\xrightarrow{!}$.

$$
\begin{array}{rcl}
\downarrow^{\bullet} e & = & e \\
\downarrow^{\sigma_1 \to \sigma_2} f & = & \underline{\lambda} x.\mathsf{reset}\langle \downarrow^{\sigma_2} (f(\uparrow_{\sigma_1} x)) \rangle \quad (x \text{ is fresh}) \\
\downarrow^{\sigma_1 \xrightarrow{!} \sigma_2} f & = & \underline{\lambda} x.\mathsf{reset}\langle \downarrow^{\sigma_2} (f(\uparrow_{\sigma_1} x)) \rangle \quad (x \text{ is fresh}) \\
 & & \\
\uparrow_{\bullet} e & = & e \\
\uparrow_{\sigma_1 \to \sigma_2} e & = & \lambda x.\uparrow_{\sigma_2} (e \underline{@} (\downarrow^{\sigma_1} x)) \\
\uparrow_{\sigma_1 \xrightarrow{!} \sigma_2} e & = & \lambda x.\mathsf{shift}\,(\lambda k.\underline{\mathbf{let}}\ x' \Leftarrow e \underline{@} \downarrow^{\sigma_1} x\ \underline{\mathbf{in}}\ \mathsf{reset}\langle k(\uparrow_{\sigma_2} x') \rangle) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad (x' \text{ is fresh})
\end{array}
$$

Figure 17.9: TDPE with let-insertion

17.5.2 Monomorphizing control operators

In the definition of reflection $\uparrow_{\sigma_1 \xrightarrow{!} \sigma_2}$ for function types with latent effects, the return type (here σ_2) of the shift-expression *depends* on the type of the reflection. Hence it is not immediately amenable to be treated by TDPE itself, because during self-application, shift is regarded as a dynamic constant, whose type is needed to determine its residualizing instantiation.

However, observe that the argument to the context k is fixed to be $\uparrow_{\sigma_2} x'$; this prompts us to move this term into the context surrounding the shift-expression, and to apply k to a simple unit value $()$. Following this transformation, no information needs to be carried around, except for the transfer of the control flow.

$$\uparrow^{\mathrm{new}}_{\sigma_1 \xrightarrow{!} \sigma_2} \; e = \lambda x.(\lambda\,().\uparrow_{\sigma_2} x')(\mathsf{shift}\,(\lambda k.\underline{\mathbf{let}}\; x' \Leftarrow e\underline{@}{\downarrow}^{\sigma_1}\; x\; \underline{\mathbf{in}}\; \mathsf{reset}\langle k()\rangle))$$

(x' is fresh)

Now the aforementioned problem is solved, since the return type of shift is fixed to unit—the new definition is *monomorphic*.

To show that this change is semantics-preserving, we compare the CPS semantics of the original definition and the new definition.

Proposition 17.3. *The terms* $[\![\uparrow^{\mathrm{new}}_{\sigma_1 \xrightarrow{!} \sigma_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\sigma_1 \xrightarrow{!} \sigma_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Here β_v and η_v are respectively the β and η rules in Moggi's computational lambda calculus λ_c [74], i.e., the restricted forms of the usual β rule, $(\lambda x.e')e \sim e'\{e/x\}$, and of the usual η rule, $\lambda x.ex \sim e$, where the expression e must be a value. These rules are sound for call-by-value languages with computational effects.

Proof. First of all, we abstract out the same computations in the two terms:

$$\begin{aligned} B &\equiv \lambda f.\underline{\mathbf{let}}\; x' \Leftarrow e\underline{@}{\downarrow}^{\sigma_1}\; x\; \underline{\mathbf{in}}\; f() \\ R &\equiv \uparrow_{\sigma_2} x' \\ C[\,] &\equiv \lambda e.\lambda x.\mathbf{let}\; x' \Leftarrow \mathsf{new}()\; \mathbf{in}\; [\,] \end{aligned}$$

Then

$$\uparrow_{\sigma_1 \overset{!}{\to} \sigma_2} =_{\beta_v\eta_v} C[\mathsf{shift}\,(\lambda\, k.B(\lambda().\mathsf{reset}\langle k(R)\rangle))]$$

$$\uparrow^{\mathrm{new}}_{\sigma_1 \overset{!}{\to} \sigma_2} =_{\beta_v\eta_v} C[(\lambda().R)(\mathsf{shift}\,(\lambda\, k.B(\lambda().\mathsf{reset}\langle k()\rangle)))]$$

Because the CPS transformation is compositional and preserves $\beta_v\eta_v$ equivalence, it suffices to prove that the CPS transformations of the two terms enclosed by $C[\cdot]$ are $\beta_v\eta_v$-equivalent, for all terms B and R. It is a tedious but straightforward check. □

Recently, Sumii [96] pointed out that the reset in the above definition can be removed. The continuation k, being captured by shift, resets the continuation automatically when applied to an argument, which makes the reset in the above definition redundant—this is because, in particular, the argument of k is a *value*. In contrast, the original definition still requires the reset, since the expression $\uparrow_{\sigma_2} x'$ might have latent escaping control effect, as in the case where σ_2 is of form $\sigma \overset{!}{\to} \sigma'$. This simplification improves the performance of TDPE and the generating extension generated by self-application.

$$\uparrow^{\mathrm{new}'}_{\sigma_1 \overset{!}{\to} \sigma_2} \quad e = \lambda\, x.(\lambda\,().\uparrow_{\sigma_2} x')(\mathsf{shift}\,(\lambda\, k.\underline{\mathbf{let}}\ x' \Leftarrow e\,\underline{@}{\downarrow}^{\sigma_1}\ x\ \underline{\mathbf{in}}\ k()))$$

(x' is fresh)

Proposition 17.4. *The terms* $[\![\uparrow^{\mathrm{new}}_{\sigma_1 \overset{!}{\to} \sigma_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow^{\mathrm{new}'}_{\sigma_1 \overset{!}{\to} \sigma_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Proof. We proceed as in the proof of Proposition 17.3. In particular, using B, R, and $C[\,]$ introduced there, we have that

$$\uparrow^{\mathrm{new}'}_{\sigma_1 \overset{!}{\to} \sigma_2} =_{\beta_v\eta_v} C[(\lambda().R)(\mathsf{shift}\,(\lambda\, k.B(\lambda().k())))].$$

□

Example 17.5. *The monomorphic definitions* $\uparrow^{\text{new}}_{\sigma_1 \xrightarrow{!} \sigma_2}$ *and* $\uparrow^{\text{new}'}_{\sigma_1 \xrightarrow{!} \sigma_2}$ *of reflection for function types with latent effects are amenable to TDPE itself. Figure 17.10 shows the result of visualizing the reification function at the type* $(\bullet \xrightarrow{!} \bullet) \xrightarrow{!} \bullet$. *Note that both* shift *and* reset *have effects themselves; consequently TDPE has inserted let-constructs for the result of visualization. For comparison, we also show the visualization of* $(\bullet \to \bullet) \to \bullet$ *of Pure NbE, which is much more compact.*

The main difference here is the control operators used in Full TDPE, which remain in the result of self-application; later in Chapter 17.7, we will see how this difference affects the speedup achieved by the second Futamura projection.

17.5.3 Sum types

Full TDPE also treats sum types using control operators; this treatment is also due to Danvy [12]. Briefly, the operator shift is used in the definition of reflection function for sum types, $\uparrow_{\sigma_1+\sigma_2}$. As the type suggests, the return type of this function should be a value of type $\overline{\sigma_1} + \overline{\sigma_2}$, i.e., a value either of the form **inl** $(v_1 : \overline{\sigma_1})$ or **inr** $(v_2 : \overline{\sigma_2})$ (for some appropriate v_1 or v_2); on the other hand, both values are needed to have the complete information. Danvy's solution is to "return twice" to the context by capturing the delimited context and applying it separately to **inl** $(\uparrow_{\sigma_1} e_1)$ and **inr** $(\uparrow_{\sigma_2} e_2)$; the results are combined using a case-construct which introduces the bindings for e_1 and e_2. Danvy's definition

Visualization of $\downarrow^{(\bullet \stackrel{!}{\to} \bullet) \stackrel{!}{\to} \bullet}$ results in:

$\lambda x_1.$**let** $r_2 = \mathsf{init}()$

$\quad r_3 = \mathsf{new}()$

$\quad r_{11} = \mathsf{reset}\langle$**let** $r_{10} = x_1(\lambda x_5.\mathsf{insertLet}(\underline{r_3}\,\underline{@}\,x_5))$ **in** r_{10} **end**$\rangle$

in $\underline{\lambda} r_3.r_{11}$

end

where $\mathsf{insertLet}(E)$ abbreviates the expression

let $x' = \mathsf{new}()$

$\quad \_ = \mathsf{shift}(\lambda k.$**let** $r = k()$ **in** $(\underline{\textbf{let}}\ x' \Leftarrow E\ \underline{\textbf{in}}\ r)$ **end**$)$

in $\underline{x'}$

end

In contrast, visualization of $\downarrow^{(\bullet \to \bullet) \to \bullet}$ of Pure NbE results in:

$\lambda x_1.$**let** $r_2 = \mathsf{init}()$

$\quad r_3 = \mathsf{new}()$

in $\underline{\lambda} r_3.x_1(\lambda x_4.\underline{r_3}\,\underline{@}\,x_4)$

end

Figure 17.10: Visualizing TDPE with let-insertion

of $\uparrow_{\sigma_1+\sigma_2}$ is given below:

$$\begin{aligned}\uparrow_{\sigma_1+\sigma_2} e = \mathsf{shift}(\lambda k.\ \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \underline{\mathbf{inl}}(\underline{x_1}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inl}\ (\uparrow_{\sigma_1}\ \underline{x_1}))\rangle \\ \mid\ \underline{\mathbf{inr}}(\underline{x_2}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inr}\ (\uparrow_{\sigma_2}\ \underline{x_2}))\rangle\ ) \\ (x_1, x_2 \text{ are fresh})\end{aligned}$$

where `Exp` has been extended with constructors for a case distinction and injection functions in the obvious way. Again, the return type of the shift-expression in the above definition is not fixed; an alternative definition is needed to allow self-application.

Following the same analysis as before, we observe that the arguments to k must be one of the two possibilities, $\mathbf{inl}\ (\uparrow_{\sigma_1}\ e_1)$ and $\mathbf{inr}\ (\uparrow_{\sigma_2}\ e_2)$, so the information to be passed through the continuation is just the binary choice between the left branch and the right branch. We can thus move these two fixed arguments into the context and replace them with the booleans tt and ff as the argument to continuation k (again, Sumii's remark on the redundancy of reset in the program after change applies, and we have dropped the unnecessary occurrences of reset):

$$\begin{aligned}\uparrow^{\mathrm{new}}_{\sigma_1+\sigma_2} e = \ & \mathbf{if}\ \mathsf{shift}(\lambda k.\ \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ \underline{\mathbf{inl}}(\underline{x_1}) \Rightarrow k\ \mathsf{tt} \\ & \qquad\qquad \mid\ \underline{\mathbf{inr}}(\underline{x_2}) \Rightarrow k\ \mathsf{ff}\ ) \\ & \mathbf{then\ inl}\ (\uparrow_{\sigma_1}\ \underline{x_1})\ \mathbf{else\ inr}\ (\uparrow_{\sigma_2}\ \underline{x_2}) \\ & \qquad\qquad (x_1, x_2 \text{ are fresh})\end{aligned}$$

The use of shift is instantiated with the fixed boolean type. Again, we check that this change does not modify the semantics.

Proposition 17.6. $[\![\uparrow^{\mathrm{new}}_{\sigma_1+\sigma_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\sigma_1+\sigma_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Using $\uparrow^{\text{new}}_{\sigma_1+\sigma_2}$ and $\uparrow^{\text{new}'}_{\sigma_1 \overset{!}{\to} \sigma_2}$ instead of the original definitions provides us with an algorithm for Full TDPE that is amenable to self-application. In the following section, we use self-application of Full TDPE for compiler generation.

17.6 An application: generating a compiler for Tiny

It is well known that partial evaluation allows compilation by specializing an interpreter with respect to a source program. TDPE has been used for this purpose in several instances [11, 12, 22, 24]. Having implemented the second Futamura projection, we can instead *generate* a compiler as the generating extension of an interpreter.

One of the languages for which compilation with TDPE has been studied is *Tiny* [11, 84], a prototypical imperative language. As outlined in Section 15.2, a functor `tiny_pe(D:DYNAMIC)` is used to carry out type-directed partial evaluation in a convenient way. This functor provides an interpreter `meaning` that is parameterized over all dynamic constructs. Appendix E.1 gives an overview of Tiny and type-directed partial evaluation of a Tiny interpreter. Compiling Tiny programs by partially evaluating the interpreter `meaning` corresponds to running the trivial generating extension `meaning`†.

Following the development in Section 17.3, we proceed in three steps to generate a Tiny compiler:

1. Rewrite `tiny_pe` into a different functor `tiny_ge(S: STATIC D: DYNAMIC)` in which `meaning` is also parameterized over all static constants and base types.

2. Give instantiations of `S` and `D` as indicated by the instantiation table in

Table 17.1 (page 184), thereby creating the GE-instantiation "`meaning`".

3. Perform the second Futamura projection; this yields the efficient generating extension `meaning`‡, i.e., a Tiny compiler.

Appendix E.2 describes these steps in more detail.

Tiny was the first substantial example we treated; nevertheless we were done within a day—none of the three steps described above is conceptually difficult. They can be seen as a methodology for performing the second Futamura projection in TDPE on a binding-time-separated program.

Although conceptually simple, the first of the three aforementioned steps is somewhat tedious:

- Every construct that is not handled automatically by TDPE has to be parameterized over. This is not a problem for user-defined constants, but is a problem for ML-constructs like recursion and case-distinctions over recursive data types. Both have to be rewritten, using fixed-point operators and elimination functions, respectively.

- For every occurrence of a constant in the program, its monotype has to be determined; constants used at more than one monotype give rise to several instances. This is a consequence of performing *type-directed* partial evaluation; for the second Futamura projection, every constant is instantiated with a code-generation function, the form of which depends on the exact type of the constant in question.

Because the Tiny interpreter we started with was already binding-time sepa-

rated, we did not have to perform the binding-time analysis needed when starting from scratch. Our experience with TDPE, however, shows that performing such a binding-time analysis is relatively easy, because

- TDPE restricts the number of constructs that have to be considered, since functions, products and sums do not require binding-time annotations, and
- TDPE uses the ML type system: Type checking checks the consistency of the binding-time annotations.

17.7 Benchmarks

17.7.1 Experiments and results

In Chapter 16 we claimed that the specialized generating extension $p^{\ddagger}$ of a program p produced by the second Futamura projection for TDPE is, in general, more efficient than the trivial generating extension $p^{\dagger}$. In order to assess how much more efficient $p^{\ddagger}$ is than $p^{\dagger}$, we performed benchmarks for $+_{\mathsf{ch}}$ (Section 17.2) and the Tiny interpreter (Chapter 17.6).

The benchmarks were performed on a 250 MHz Silicon Graphics O_2 workstation using Standard ML of New Jersey version 110.0.3. We display the results in Table 17.2. In each row of the table, we compare the time it takes to specialize the subject program p with respect to the static input s using two different generating extensions: (1) the trivial generating extension $p^{\dagger}$ (i.e., directly running TDPE on program p), and (2) the specialized generating extension $p^{\ddagger}$ (i.e., running the result of the second Futamura projection). We calculate the speedup

program p	static inp. s	specialization time with $p^\dagger$ (s)	specialization time with $p^\ddagger$ (s)	Speedup (ratio)
`meaning`	`factorial`	261.2	194.9	1.34
$\texttt{meaning}_{\mathsf{orig}}$	`factorial`	169.5	99.2	1.71
$+_{\mathsf{ch}}$	80_{ch}	58.45	19.95	2.93

Table 17.2: Benchmarks: time of specializations (1,000,000 repeated executions)

as the ratio of their running times.

The first row compares the compilers derived from the interpreter `meaning` (see Chapter 17.6 and Appendix E); the result shows a speedup of 1.34 for compiling the factorial function. One might wonder, however, whether there is any real gain in using the second Futamura projection: The changes that are necessary to provide the GE-instantiation of `meaning` (replace built-in pattern-matching and recursive function definition of ML with user-defined fixed-point operators and case operators, respectively—see Chapter 17.6) slow down both direct compilation with TDPE and compilation using the specialized generating extension. In fact, as the table's second row shows, direct compilation with the 'original' interpreter $\texttt{meaning}_{\mathsf{orig}}$, i.e., an instantiation of `tiny_pe` rather than `tiny_ge` (cf. Sections 15.2 and 17.3), runs even faster than the specialized generating extension $\texttt{meaning}^\ddagger$.

We can do better by replacing the user-defined fixed point operators and case operators in the result program $\texttt{meaning}^\ddagger$ with the built-in constructs.[2] This

[2]Removing the user-defined fixed point operator and case operators can be carried out automatically by (1) incorporating TDPE with patterns as generated bindings, and (2) systematically changing

yields a program that can be understood as the specialized generating extension of the program $\texttt{meaning}_{\mathsf{orig}}$, and we thus call it $\texttt{meaning}_{\mathsf{orig}}{}^{\ddagger}$. The second row of Table 17.2 shows that running $\texttt{meaning}_{\mathsf{orig}}{}^{\ddagger}$ gives a speedup of 1.71 over running the original program $\texttt{meaning}_{\mathsf{orig}}$. The speedup over the direct compilation using the original interpreter here is, in practice, more relevant than the speedup of the benchmark shown in the first row.

The benchmark in the third row compares the generating extensions of an effect-free function, the addition function $+_{\mathsf{ch}}$ for Church numerals. Because the function is free of computational effects (we assume that its argument function is also effect-free), we can specialize Pure TDPE instead of Full TDPE in the second Futamura projection. The speedup of running the specialized generating extension over direct partial evaluation is consistently around 3 (shown with Church numeral 80_{ch}).

17.7.2 Analysis of the result

Overall, the speedup of the second Futamura projection with TDPE is disappointing compared to the typical order-of-magnitude speedup achievable in traditional partial evaluation [57]. This, on the other hand, reflects the high efficiency of TDPE, which carries out static computations by evaluation rather than symbolic manipulation. Turning symbolic manipulation (i.e., interpretation) into evaluation is one of the main goals one hopes to achieve by specializing a syntax-directed partial evaluator. Since TDPE does not have much interpretive

the residualizing instantiations for the fixed point and case operators used. Danvy and Rhiger [22] achieved a similar effect in TDPE for Scheme, using Scheme macros.

overhead in the first place, the speedup is bound to be lower.

Logically, the next question to ask—for a better understanding of how and when the second Futamura projection could effectively speedup the process of TDPE—is what cost of TDPE can or cannot be removed by using self-application. The higher-order nature of the TDPE algorithm blurs the boundaries between the various components that contribute to the running time of the specialization; we can only roughly divide the cost involved in performing TDPE as follows:

1. Cost due to computation in the extraction function $\downarrow^{\sigma}$, namely function invocations of reification and reflection for subtypes of σ, name and code generation and, in the case of Full TDPE, the use of control operators shift and reset.

2. Cost due to computation in the residualizing instantiation $\ulcorner ps \urcorner$ of input program and static input, namely, apart from static computation, the invocation of reflection by code-generation versions of dynamic constants.

3. Cost due to reducing extra redexes formed by the interaction of $\downarrow^{\sigma}$ and $\ulcorner ps \urcorner$ in $\downarrow^{\sigma} \ulcorner ps \urcorner$.

Of the costs due to computation in the extraction function, only the one caused by function invocations can be eliminated, which amounts to function inlining. All other computations have to be performed at specialization time. Similarly, for the cost associated with the residualizing instantiation, inlining can be performed for the code-generation versions of dynamic constants and their calls to the reflection function. Finally, the extra redexes formed by the

interaction of the extraction function and the residualizing instantiation can be partly reduced by the specialization.

In Full TDPE, the somewhat time-consuming control operators dominate the cost of extraction algorithm; the low speedup of specializing Full TDPE (the first two benchmarks) as opposed to that of specializing Pure TDPE (the third benchmark), we think, are mainly due to the fact that these control operators cannot be eliminated. Furthermore, in the case of the Church addition function, the program is a higher-order pure λ-term, which usually "mixes well" with the extraction function, in the sense that many extra redexes are formed by their interaction.

Do certain implementation-related factors, such as the global optimizations of the ML compiler we used and the fact that we are working in a typed setting, give positive contribution to the speedup? In our opinion, the help is minimal, if not negative. First, the specialization carried out by the self-application with respect to a trivial BTA (Chapter 16.1) has an effect similar to a good global inliner. Therefore, the global optimization of the ML compiler, especially the inlining optimization, should only reduce the potential speedup of the specialization. Second, working in a typed setting does complicates the type specification and the parameterization (Section 17.4), but it does not incur extra cost at runtime when using TDPE. Indeed, the instantiation through ML functors happens at compile time. Furthermore, the need to parameterize over built-in constructs such as fixed point operators and pattern matching is present also in an untyped setting.

Chapter 18

Concluding remarks for Part III

We have adapted the underlying concept of the second Futamura projection to TDPE and derived an ML implementation for it. By treating several examples, among them the generation of a compiler from an interpreter, we have examined the practical issues involved in using our implementation for deriving generating extensions of programs.

To build a *generating-extension generator* (cogen), and to formally prove its correctness at the same time, one possibility is to start with a partial evaluator and rewrite it into the desired generating extension in several steps, such as the use of higher-order abstract syntax and deforestation in Thiemann's work [101]. Correctness follows from showing the correctness of the partial evaluator and the correctness of each of these steps. In contrast, for generating extensions produced with the second Futamura projection, the implementation is produced automatically, and correctness follows immediately from the correctness of the partial evaluator. Often, however, this conceptual simplicity is compromised by

(1) the complications in using self-application, and (2) the need to make the partial evaluator self-applicable and prove the necessary changes to be meaning preserving. In the case of TDPE, the implementation effort for writing the GE-instantiation of the object program is similar in level to that of the hand-written cogen approach, but the only change to the TDPE algorithm itself is the transformation described and proven correct in Section 17.5.

The third Futamura projection states that specializing a partial evaluator with respect to itself yields an efficient generating-extension generator. The type-indexed nature of TDPE makes it challenging to implement the third Futamura projection directly in ML. If it can be done, our experience with the second Futamura projection suggests that only an insignificant speedup would be obtained.

At the current stage, our contribution seems to be more significant at a conceptual level, since the speedup achieved by using the generated generating extensions is rather modest. However we observed that a higher speedup can be achieved for more complicated type structures, especially in a setting with no or few uses of computational effects; this suggests that our approach to the second Futamura projection using TDPE might find more practical applications in, e.g., the field of type theory and theorem proving.

The technical inconveniences mentioned in Chapter 17.6 are clearly an obstacle for using the second Futamura projection for TDPE (and, to a lesser extent, for using TDPE itself). A possible solution is to implement a translator from the two-level language into ML, thus handling the mentioned technicalities automatically. Of course, such an approach would sacrifice the flexibility of TDPE

of allowing the use of all language constructs in the static part of the subject program. Even so, TDPE would still retain a distinct flavor when compared to traditional partial-evaluation techniques: Only those constructs not handled automatically by TDPE, i.e., constants, need to be binding-time annotated; other constructs, such as function application and function abstraction, always follow their standard typing rules from typed lambda calculi. This simplifies the binding-time analysis considerably and often makes binding-time improvements, e.g, eta-expansion, unnecessary, which was one of the original motivations of TDPE [12, 21].

Appendices

Appendix A

Call-by-name CPS translation

Danvy and Filinski [18] also presented a one-pass version for Plotkin's call-by-name CPS transformation. Figure A.1 shows both transformations. The erasure argument applies here, too.

Other one-pass CPS transformations [43] can be similarly described and proven correct.

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

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

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

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

b. Plotkin's original transformation:

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

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

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

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

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

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

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

Figure A.1: Call-by-name CPS transformation (1/2)

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

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

1. The (higher-order) auxiliary translation

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

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

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

2. The complete translation

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

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

Figure A.1: Call-by-name CPS transformation (2/2)

Appendix B

Expanded proofs for nPCF^2

B.1 Type preservation and annotation erasure

Theorem 4.1 (Type preservation). *If* $\bigcirc\Delta \rhd E : \tau$ *and* $E \Downarrow V$, *then* $\bigcirc\Delta \rhd V : \tau$.

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

Theorem 4.3 (Annotation erasure). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$, *then* $\mathsf{nPCF} \vdash \Delta \rhd |E| = |V| : |\tau|$.

Its proof uses the following Substitution Lemma for erasure.

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

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

Proof of Theorem 4.3. By rule induction on $E \Downarrow V$. We show a few cases.

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

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

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

B.2 Native embedding

We first present the standard denotational semantics of the language $\mathsf{nPCF}^\wedge$.

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

The standard domain-theoretical semantics maps $\mathsf{nPCF}^\wedge$*-types to domains as follows.*

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

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

continuous function $[\![E]\!] : [\![\Delta]\!] \to [\![\sigma]\!]$*:*

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

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

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

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

terminates.

Lemma 4.6 (Substitution lemma for $\{\!|-|\!\}_{\mathsf{n}\epsilon}$). *If* $\mathsf{nPCF}^2 \vdash \Gamma, x : \tau' \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash \Gamma \rhd E' : \tau'$, *then* $\{\!|E\{E'/x\}|\!\}_{\mathsf{n}\epsilon} \sim_\alpha \{\!|E|\!\}_{\mathsf{n}\epsilon}\{\{\!|E'|\!\}_{\mathsf{n}\epsilon}/x\}$.

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

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

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

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

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

□

Lemma 4.7 (Evaluation preserves translation). *If* $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$, *then* $\mathsf{nPCF}^\Lambda \vdash \{\!|\bigcirc\Delta|\!\}_{\mathsf{n}\epsilon} \rhd \{\!|E|\!\}_{\mathsf{n}\epsilon} = \{\!|V|\!\}_{\mathsf{n}\epsilon} : \{\!|\tau|\!\}_{\mathsf{n}\epsilon}$.

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

Case [*app*]: $\{\!|E_1\,E_2|\!\}_{\mathsf{n}\epsilon} \equiv \{\!|E_1|\!\}_{\mathsf{n}\epsilon}\,\{\!|E_2|\!\}_{\mathsf{n}\epsilon}$

$$\begin{aligned}
&\overset{i.h.}{=} \{\!|\lambda x.E'|\!\}_{\mathsf{n}\epsilon}\{\!|E_2|\!\}_{\mathsf{n}\epsilon} \equiv (\lambda x.\{\!|E'|\!\}_{\mathsf{n}\epsilon})\,\{\!|E_2|\!\}_{\mathsf{n}\epsilon} \\
&= \{\!|E'|\!\}_{\mathsf{n}\epsilon}\{\{\!|E_2|\!\}_{\mathsf{n}\epsilon}/x\} && (\beta) \\
&\sim_\alpha \{\!|E'\{E_2/x\}|\!\}_{\mathsf{n}\epsilon} && (\textit{Lemma 4.6}) \\
&\overset{i.h.}{=} \{\!|V|\!\}_{\mathsf{n}\epsilon}.
\end{aligned}$$

Case [*fix*]: $\{\!|\mathbf{fix}\,E|\!\}_{\mathsf{n}\epsilon} \equiv \mathbf{fix}\,\{\!|E|\!\}_{\mathsf{n}\epsilon}$

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

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

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

Lemma 4.8 (Translation of code-typed value). *If* $\mathsf{nPCF}^2 \vdash v_1 : \bigcirc\sigma_1, \ldots, v_n : \bigcirc\sigma_n \rhd \mathcal{O} : \bigcirc\sigma$, *then there is a value* $t : \Lambda$ *such that*

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

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

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

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

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

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

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

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

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

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

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

Lemma 4.9 (Computational adequacy). *If* $\mathsf{nPCF}^2 \vdash \rhd E : \bigcirc\sigma$*, and there is a* nPCF^Λ*-value* $t : \Lambda$ *such that* $[\![\{\!|E|\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$*, then* $\exists \mathcal{O}.E \Downarrow \mathcal{O}$.

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

We prove Lemma 4.9 using a Kripke logical relation between the denotation of translated terms and the original two-level terms.

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

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

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

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

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

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

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

The denotational semantics forces us to take a standard call-by-name form of logical relation at the function type in the definition. Another problem then appears: since we do not evaluate the expression form in the operational semantics, what we can infer from the induction hypothesis does not directly give the conclusion. In particular, for the case $\lambda x.E$, using the induction hypothesis, we can relate the denotation to the expression $E\{\theta; E'/x\}$, but instead we need to relate it to the expression $((\lambda x.E)\{\theta\})E'$. The two expressions evaluate to the same value in the operational semantics, i.e., they are Kleene-equivalent. Therefore, we need to show that the logical relation can be transferred between Kleene-equivalent terms.[1]

[1] In the literature, this mismatch problem is resolved by using a different formulation of the logical relation, usually called *computability*: the relation at higher types is defined by means of full applications (e.g., Plotkin's proof of adequacy [88] and Gunter's proof [40, Section 4.3]), which reduces the definition at higher-type directly to ground type. The Kleene-equivalence formulation used in the present article has the same effect, but it seems to scale better with respect to other types such as

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

Proof. By a case analysis on type τ.

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

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

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

Proof. By induction on type τ.

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

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

product and sum; the definition of full applications becomes hairy in the presence of these types.

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

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

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

It is admissible since all the $- \prec_{\tau_2}^{\Delta} EE'$ are admissible by induction hypothesis, and admissibility is closed under taking pre-image under strict continuous function and arbitrary intersection [107]. □

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

Proof. By induction on type τ.

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

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

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

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

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

The logical relation extends naturally from types to typing contexts. Let Γ be a nPCF^2-typing context $(x_1 : \tau_1, \ldots, x_n : \tau_n)$, Δ an object-type typing context,

and θ a substitution (from $\bigcirc\Delta$ to Γ) $\{E_1/x_1, \ldots, E_n/x_n\}$, where $\mathsf{nPCF}^2 \vdash \bigcirc\Delta \rhd E_i : \tau_i$. We define the relation $\prec^{\Delta}_{\Gamma}$ between environments and substitutions as follows:

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

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

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

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

The static term formations have fairly standard sub-proofs.

Case ℓ**:** Trivial.

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

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

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

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

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

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

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

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

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

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

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

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

There are three sub-cases:

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

Case $E_1 \otimes E_2$: Simple.

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

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

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

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

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

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

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

Then, by the induction hypothesis, we get

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

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

Finally, Lemma 4.9 is an easy corollary.

Proof of Lemma 4.9. Let ρ be the empty environment, and θ the empty substitution. We have that $[\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}(1)]\!]\rho = [\![t]\!]\rho$. Since t is a value, $[\![t]\!]\rho \neq \perp$. Thus $[\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}]\!]\rho(\mathbf{val}^{\perp}\,1) = [\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}(1)]\!]\rho \neq \perp$. By Lemma B.8, $[\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}]\!]\rho \prec^{\cdot}_{\bigcirc\sigma} E\{\theta\} \equiv E$. The definition of the logical relation at the type $\bigcirc\sigma$ implies that $E \Downarrow$. □

Theorem 4.10 (Correctness of embedding). *If* $\mathsf{nPCF}^{2} \vdash \triangleright E : \bigcirc\sigma$, *then the following statements are equivalent.*

(a) There is a value $\mathcal{O} : \bigcirc\sigma$ *such that* $\mathsf{nPCF}^{2} \vdash E \Downarrow \mathcal{O}$.

(b) There is a value $t : \Lambda$ *such that* $[\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t]\!]$.

When these statements hold, we further have that

(c) $\mathsf{nPCF} \vdash \triangleright\mathcal{D}(t) : \sigma$ *and* $|\mathcal{O}| \sim_{\alpha} \mathcal{D}(t)$.

Proof. (a) ⇒ (b),(c) Assume (a). By Lemma 4.7, $\mathsf{nPCF}^{\Lambda} \vdash \{\!\{E\}\!\}_{\mathsf{n}\epsilon} = \{\!\{\mathcal{O}\}\!\}_{\mathsf{n}\epsilon}$. By Lemma 4.8, there is a value t such that $\mathsf{nPCF}^{\Lambda} \vdash \{\!\{\mathcal{O}\}\!\}_{\mathsf{n}\epsilon}1 = t$, with which (c) also holds. By the transitivity rule, we have that $\mathsf{nPCF}^{\Lambda} \vdash \{\!\{E\}\!\}_{\mathsf{n}\epsilon}1 = t$. The conclusion now follows by an application of Theorem B.3.

(b) ⇒ (a),(c) Assume (a). By Lemma 4.9, $E \Downarrow$. Since $\triangleright E : \bigcirc\sigma$, we have $E \Downarrow \mathcal{O}$ for some value $\mathcal{O} : \bigcirc\sigma$. By the first part of this proof, we further

have an nPCF^{Λ}-value $t' : \Lambda$ that satisfies (c) (with t replaced by t') and validates $[\![\{\!\{E\}\!\}_{\mathsf{n}\epsilon}(1)]\!] = [\![t']\!]$. It remains to show that $t \equiv t'$.

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

□

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

Lemma 4.11. *For all types* σ, $\mathsf{nPCF} \vdash \rhd |\Downarrow^{\sigma}| = \lambda x.x : \sigma \to \sigma$ *and* $\mathsf{nPCF} \vdash \rhd |\Uparrow_{\sigma}| = \lambda x.x : \sigma \to \sigma$.

Proof. By a straightforward induction on type τ. □

Theorem 4.12 (Semantic correctness of TDPE). *If* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$ *and* $\mathsf{nPCF}^{2} \vdash \mathsf{NF}(E) \Downarrow \mathcal{O}$, *then* $\mathsf{nPCF} \vdash \rhd |\mathcal{O}| = |E| : \sigma$.

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

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

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

Case $d^\emptyset : \sigma^\emptyset$: We have $|\{\!|d^\emptyset|\!\}_{\mathsf{ni}}| \equiv |\uparrow_\sigma||\underline{d}| \stackrel{*}{=} d \equiv |d^\emptyset|$, where $\stackrel{*}{=}$ uses Lemma 4.11 and the definition of erasure for nPCF^2-terms.

From this, we can infer that $|\mathsf{NF}(E)| \equiv |\downarrow^\sigma||\{\!|E|\!\}_{\mathsf{ni}}| = |E|$, again using Lemma 4.11. Now, applying Theorem 4.3 to $\mathsf{nPCF}^2 \vdash \mathsf{NF}(E) \Downarrow \mathcal{O}$, we can conclude that $|\mathcal{O}| = |\mathsf{NF}(E)| = |E|$. □

Theorem 4.13 (Refined type preservation). *If* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright E : \tau$ *and* $\mathsf{nPCF}^2 \vdash E \Downarrow V$*, then* $\mathsf{nPCF}^2 \vdash \bigcirc^{var}(\Delta) \blacktriangleright V : \tau$.

Like in the proof of Theorem 4.1, the most interesting case is when $E \equiv E_1\, E_2$, for which we need a Substitution Lemma.

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

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

Proof of Theorem 4.13. By induction on $E \Downarrow V$. The only non-straightforward case in the static part is the rule ([*app*]), i.e., the evaluation of an application; for this rule we use Lemma B.9. For the dynamic part, all the rules simply evaluate the subterms while keeping the top-level constructs; combining the induction hypotheses suffices to give the same typing for the result values V as for the original terms E. □

Theorem 4.14 (Normal-form code types). *If* V *is an* nPCF^2*-value (Figure 4.2), then*

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

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

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

Lemma 4.15. *(1) The extraction functions (Figure 4.5c) have the following normal-form types (writing* $\sigma^{\bigcirc\mathsf{nf}}$ *for* $\sigma\{\bigcirc^{nf}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*).*

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

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

Proof.

(1) By induction on type σ.

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

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

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

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

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

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

Theorem 4.16. *If* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, *then* $\mathsf{nPCF}^{2} \vdash\blacktriangleright \mathsf{NF}(E) : \bigcirc^{nf}(\sigma)$.

Proof. By Lemma 4.15(2), we have $\mathsf{nPCF}^{2} \vdash\blacktriangleright \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\sigma^{\mathfrak{d}}|\!\}_{\mathsf{ri}}^{\mathsf{nf}}$. Since $\{\!|\sigma^{\mathfrak{d}}|\!\}_{\mathsf{ri}}^{\mathsf{nf}} \equiv \sigma^{\bigcirc\mathsf{nf}}$, applying $\Downarrow^{\sigma} : \sigma^{\bigcirc\mathsf{nf}} \to \bigcirc^{nf}(\sigma)$ (Lemma 4.15(1)) to $\{\!|E|\!\}_{\mathsf{ri}}$ yields the conclusion. □

Corollary 4.17 (Syntactic correctness of TDPE). *For* $\mathsf{nPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, *if* $\mathsf{nPCF}^{2} \vdash \mathsf{NF}(E) \Downarrow V$, *then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\mathsf{nPCF} \vdash \Delta \rhd^{nf} |\mathcal{O}| : \sigma$.

Proof. We use Theorem 4.16, Theorem 4.13, and Theorem 4.14. □

Appendix C

Call-by-value two-level language vPCF²: detailed development

Since we are working with vPCF² in this section, we leave vPCF² $\vdash$ implicit.

C.1 Type preservation

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

For a concise presentation, we introduce the following notations.

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

Definition C.3 (Binder extension and difference). *We write* $\Gamma \rhd [B] \geq [B']$, *if binder* B' *is a prefix of binder* B, *and* $\Gamma \rhd [B]$. *In this case, it makes sense*

to write $B - B'$ *to denote the difference of the two binders, and we have that* $\Gamma, ⓥ B \rhd [B - B']$.

We can restate Definition 5.1 using the preceding notions.

Definition 5.1 (Binder-term-in-context). *We write* $\Gamma \rhd [B]E : \tau$ *if* $\Gamma \rhd [B]$ *and* $\Gamma, ⓥ B \rhd E : \tau$.

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

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

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

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

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

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

(1.b) $\Gamma, ⓥB \rhd E' : \tau_1$.

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

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

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

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

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

(1) $\Gamma, ⓥB \rhd E : \tau$.

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

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

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

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

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

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

We slightly strengthen the type preservation theorem 5.2, so as to make the induction go through.

Theorem 5.2 (Type preservation). *If* $\textcircled{v}\Delta \rhd [B]E : \tau$ *and* $\Delta \rhd [B]E \Downarrow [B']V$, *then (1)* $\textcircled{v}\Delta \rhd [B']V : \tau$, *and (2)* $\textcircled{v}\Delta \rhd [B'] \geq [B]$.

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

For (1), we show a few cases.

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

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

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

for which we reason as follows:

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

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

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

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

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

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

   b. $\textcircled{v}\Delta \rhd [B_2] \geq [B_1]$.

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

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

By induction hypothesis 2,

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

b. $\textcircled{v}\Delta \rhd [B_3] \geq [B_2]$.

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

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

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

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

By induction hypothesis 3,

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

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

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

for which we reason as follows:

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

(2) Inverting (1) gives

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

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

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

(4) Inverting (3) gives

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

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

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

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

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

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

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

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

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

for which we reason as follows:

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

(2) Inverting (1) gives

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

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

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

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

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

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

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

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

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

Case [*let*]**:** Similar to the case of rule [*lam*].

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

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

for which we reason as follows:

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

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

(2) Inverting (1) gives

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

(3) Use induction hypothesis on (2) gives

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

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

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

□

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

Definition 5.3 (Observation of complete program). *For a complete program $\rhd E : \textcircled{e}\sigma$, we write $E \searrow \underline{\mathbf{let}}^*\ B\ \underline{\mathbf{in}}\ \mathcal{O}$ if $\rhd[\cdot]E \Downarrow [B]\mathcal{O}$.*

In accordance with this understanding of complete-program semantics, the following corollary of the Type Preservation theorem (Theorem 5.2) provides type preservation for a complete program.

Corollary 5.4 (Type preservation for complete programs). *If $\rhd E : \textcircled{e}\sigma$ and $E \searrow \mathcal{O}$, then $\rhd\mathcal{O} : \textcircled{e}\sigma$.*

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

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

where $(*)$ follows from Theorem 5.2 and $(**)$ follows from Lemma C.7. □

C.1.1 Determinacy

Next, we would like to show that the operational semantics of vPCF^2 is deterministic. At the top level, determinacy can be easily phrased as "evaluating a whole program gives a unique result, modulo α-equivalence". Going into the inductive steps for the proof, however, requires an extension of the notion of α-equivalence to take into account of binders, and in turn, of contexts. This extra conceptual complexity is induced, in particular, by the explicit name generations and context manipulation in rules such as [*lam*].

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

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

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

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

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

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

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

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

Proof. Immediate. □

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

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

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

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

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

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

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

for which we have the following reasoning:

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

   a. $(\circledv\Delta \rhd [B_1]E_1) \sim_\alpha (\circledv\Delta' \rhd [B_1']E_1')$, and

   b. $(\circledv\Delta \rhd [B_1]E_2) \sim_\alpha (\circledv\Delta' \rhd [B_1']E_2')$.

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

$$(\circledv\Delta \rhd [B_2]\lambda x.E_3) \sim_\alpha (\circledv\Delta' \rhd [B_2']\lambda x.E_3').$$

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

$$(\circledv\Delta \rhd [B_3]V_1) \sim_\alpha (\circledv\Delta' \rhd [B_3']V_1')$$

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

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

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

for which we have the following reasoning:

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

$$(\circledv\Delta, \circledv B_1 \rhd \lambda x.E) \sim_\alpha (\circledv\Delta', \circledv B_1' \rhd \lambda x'.E')$$

which, by definition, implies that

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

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

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

By Lemma C.11, we have

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

$$\sim_\alpha$$

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

This implies the conclusion.

□

C.2 Annotation erasure

First, we display the equational rules of vPCF in Figures C.1 and C.2. As mentioned in Chapter 5, vPCF is an instance of Moggi's computational λ-calculus [74].

Lemma C.14 (Erasure of θ-typed values). *Let V be a* vPCF^2*-value (see Figure 5.2). If $\Gamma \rhd V : \theta$ (i.e., V is of a substitution-safe type), then $|V|$ is a* vPCF*-value.*

Lemma 5.5 (Annotation erasure). *If* vPCF$^2 \vdash \circledv\Delta \rhd [B]E : \tau$ *and* vPCF$^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$, *then* vPCF$\vdash \Delta \rhd \mathbf{let}^*\ |B|\ \mathbf{in}\ |E| = \mathbf{let}^*\ |B'|\ \mathbf{in}\ |V| : |\tau|$.

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

Raw terms $E \;::=\; \ell \mid x \mid d \mid \lambda x.E \mid E_1\, E_2 \mid \mathbf{fix}\, E$

$\mid \mathbf{if}\, E_1\, E_2\, E_3 \mid E_1 \otimes E_2 \mid \mathbf{let}\; x \Leftarrow E_1\; \mathbf{in}\; E_2$

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

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

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

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

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

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

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

Figure C.1: One-level call-by-value language vPCF: syntax

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

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

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

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

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

The equational rules distinguish a subset of (possibly non-closed) terms, called *values*, ranged over by meta-variable V.

Values $V ::= \ell \mid x \mid \lambda x.E \mid d$

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

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

Equations for let-expressions

$[\mathit{unit}]$ $\quad \Delta \rhd \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ x = E : \sigma$

$[\mathit{assoc}]$ $\quad \Delta \rhd \mathbf{let}\ x_2 \Leftarrow (\mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ E_2)\ \mathbf{in}\ E$
$\qquad = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ E : \sigma$

$[\mathit{let}.\beta]$ $\quad \Delta \rhd \mathbf{let}\ x \Leftarrow V\ \mathbf{in}\ E = E\{V/x\} : \sigma$

$[\mathit{let.app}]$ $\quad \Delta \rhd E_1\, E_2 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ x_1\, x_2 : \sigma$

$[\mathit{let.fix}]$ $\quad \Delta \rhd \mathbf{fix}\, E = \mathbf{let}\ x \Leftarrow E\ \mathbf{in}\ \mathbf{fix}\, x : \sigma_1 \to \sigma_2$

$[\mathit{let.if}]$ $\quad \Delta \rhd \mathbf{if}\, E_1\, E_2\, E_3 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{if}\, x\, E_2\, E_3 : \sigma$

$[\mathit{let}.\otimes]$ $\quad \Delta \rhd E_1 \otimes E_2 = \mathbf{let}\ x_1 \Leftarrow E_1\ \mathbf{in}\ \mathbf{let}\ x_2 \Leftarrow E_2\ \mathbf{in}\ x_1 \otimes x_2 : \mathsf{b}$

Other rules

$[\beta_v]$ $\quad \Delta \rhd (\lambda x.E)V = E\{V/x\} : \sigma$

$[\eta_v]$ $\quad \Delta \rhd (\lambda x.f\, x) = f : \sigma_1 \to \sigma_2$

$[\mathit{fix\text{-}dinat}]$ $\quad \Delta \rhd \mathbf{fix}\,(f \circ g) = f(\mathbf{fix}\,(g \circ f)) : \sigma_1 \to \sigma_2$

$[\mathit{if}\text{-}\mathsf{tt}]$ $\quad \Delta \rhd \mathbf{if}\ \mathsf{tt}\, E_2\, E_3 = E_2 : \sigma$

$[\mathit{if}\text{-}\mathsf{ff}]$ $\quad \Delta \rhd \mathbf{if}\ \mathsf{ff}\, E_2\, E_3 = E_3 : \sigma$

$[\mathit{if}\text{-}\eta]$ $\quad \Delta \rhd \mathbf{if}\, x\, E\, E = E : \sigma$

$[\otimes]$ $\quad \Delta \rhd \ell_1 \otimes \ell_2 = \ell : \mathsf{b} \quad (\ell_1 \otimes \ell_2 = \ell)$

Figure C.2: One-level call-by-value language vPCF: equational theory

Case [*app*]**:** Applying Theorem 5.2 to $E_2 \Downarrow V'$, and then using Lemma C.14, we have that $|V'|$ is a vPCF-value. Now we have

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

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

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

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

Case [*lift*],[*var*],[*cst*]**:** Trivial.

Case [*lam*]**:**
$$\begin{aligned}
&\Delta; B \rhd |\underline{\lambda}x.E| \equiv \lambda x.|E| \sim_\alpha \lambda y.|E\{y/x\}| \\
\overset{i.h.}{=}\ & \lambda y.(\mathbf{let}^*\ |B'|\ \mathbf{in}\ |\mathcal{O}|) \\
\equiv\ & \mathbf{let}^*\ |[B - B]|\ \mathbf{in}\ |\lambda y.(\mathbf{let}^*\ B'\ \mathbf{in}\ \mathcal{O})|.
\end{aligned}$$

For this step we use the induction hypothesis; we also apply the congruence for

λ-abstractions.

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

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

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

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

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

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

Lemma 5.5 has the following immediate corollary for complete programs.

Theorem 5.6 (Annotation erasure for complete programs). *If* $\mathsf{vPCF}^{\mathsf{2}} \vdash \triangleright E : ⓔ\sigma$ *and* $\mathsf{vPCF}^{\mathsf{2}} \vdash E \searrow \mathcal{O}$, *then* $\mathsf{vPCF} \vdash \triangleright\,|E| = |\mathcal{O}| : \sigma$.

Proof. Assume that $\triangleright E : ⓔ\sigma$ and $E \searrow \mathcal{O}$. That is, $\triangleright[\cdot]E \Downarrow [B]\mathcal{O}'$ where $\mathcal{O} \equiv \mathbf{let}^*\ B\ \mathbf{in}\ \mathcal{O}'$). By Lemma 5.5, we have $\triangleright\,|E| = \mathbf{let}^*\ |B|\ \mathbf{in}\ |\mathcal{O}'| \equiv |\mathcal{O}| : \sigma$. □

C.3 Native implementation

C.3.1 A more "realistic" language: $\langle \mathsf{v} \rangle \mathsf{PCF}^2$

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

To establish a native implementation, we thus consider a variant of vPCF^2, $\langle \mathsf{v} \rangle \mathsf{PCF}^2$, which marks evaluated terms with angle brackets and prevents them from re-evaluation.[1] The detailed changes of $\langle \mathsf{v} \rangle \mathsf{PCF}^2$ over vPCF^2 are given in Figure C.3. Note that binders consist of only already evaluated terms, so there is no need to mark them with angle brackets.

The two languages vPCF^2 and $\langle \mathsf{v} \rangle \mathsf{PCF}^2$ are effectively equivalent, through the following operation to remove the angle brackets in $\langle \mathsf{v} \rangle \mathsf{PCF}^2$-terms:

Definition C.15 (Unbracketing). *The unbracketing of an $\langle \mathsf{v} \rangle \mathsf{PCF}^2$-term E, noted as $unbr(E)$, is the vPCF^2-term resulted from E by removing all the angle brackets in it, i.e., $unbr(\langle E \rangle) \equiv E$ and all other constructs are translated homo-*

[1] The reader might notice that angle brackets here have a similar functionality to *quote* in Lisp and Scheme. But they serve two different purposes: angle brackets here prevent re-evaluation at the semantics level, thereby removing an artifact of the substitution-based semantics and bringing the semantics closer to the actual implementation; quote in Lisp allows programmers to distinguish data from the surrounding programs. A similar concern arises when implementing, e.g., syntactic theories.

Syntax

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

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

Typing Judgment Add the following rule

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

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

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

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

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

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

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

Figure C.3: Changes of $\langle \mathsf{v} \rangle \mathsf{PCF}^2$ over vPCF^2

morphically.

We have that $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Gamma \rhd E : \tau$ *implies* $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Gamma \rhd unbr(E) : \tau$.

Theorem C.16 (Equivalence of vPCF^2 and $\langle \mathsf{v} \rangle \mathsf{PCF}^2$).

1. *If* $\mathsf{vPCF}^2 \vdash \Delta \rhd [B] unbr(E) \Downarrow [B'] V'$, *then there exists a* $\langle \mathsf{v} \rangle \mathsf{PCF}^2$*-value* V

such that $\langle \mathsf{v} \rangle \mathsf{PCF}^2 \vdash \Delta \rhd [B]E \Downarrow [B']V$ *and* $unbr(V) \equiv V'$.

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

Proof. Simple induction. □

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

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

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

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

C.3.2 The implementation language: $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$

Here we present the implementation language $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$, which can be viewed as a subset of Standard ML [71]. In detail, $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$ is vPCF with dynamic constants removed, and enriched with a global state and an inductive type Λ for representing $\langle \mathsf{v} \rangle \mathsf{PCF}^2$ code types.

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

For the state, we only model the part in which we are interested. A state S thus consists of a cell n for keeping the counter of name generation, and a cell B for keeping the accumulated bindings: $S = \langle n, B \rangle$. We also use three specialized primitives (they can be implemented using SML operations := (set) and ! (get), as shown in Figure C.4): $\mathsf{genvar}()$ generates a new variable name using the counter; $\mathsf{addBind}(E : \Lambda) : \Lambda$ adds a binding $x = E$ to the accumulated bindings such that x is a newly generated name, and returns the variable x; finally $\mathsf{letBind}(E : \Lambda)$ creates a dynamic local scope for accumulating bindings during the evaluation of term E, and inserts the accumulated bindings after the evaluation of E. The three primitives are governed by the following typing rules. The evaluation semantics of $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$ is given in Figure C.5.

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

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

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

Figure C.4: ML implementation of $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$-primitives

C.3.3 Native embedding

Definition C.18 (Embedding translation $\{\!|-|\!\}_{\mathsf{v}\epsilon}$ of $\langle\mathsf{v}\rangle\mathsf{PCF}^{2}$ into $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$).

$$\boxed{\langle\mathsf{v}\rangle\mathsf{PCF}^{2} \vdash \Gamma \rhd E : \tau} \Longrightarrow \boxed{\mathsf{vPCF}^{\Lambda,\mathfrak{st}} \vdash \{\!|\Gamma|\!\}_{\mathsf{v}\epsilon} \rhd \{\!|E|\!\}_{\mathsf{v}\epsilon} : \{\!|\tau|\!\}_{\mathsf{v}\epsilon}}$$

State $S ::= \langle n, B\rangle \qquad (n \in \mathbb{N})$

Bindings $B ::= \mathsf{nil} \mid (n, V) :: B \qquad (n \in \mathbb{N})$

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

We use the following abbreviations.

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

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

(Core)

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

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

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

(Term and State)

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

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

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

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

Figure C.5: The evaluation semantics of $\mathsf{vPCF}^{\Lambda,\mathfrak{st}}$

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

Terms : $\{\!|\$_{\mathsf{b}} E|\!\}_{\mathsf{v}\epsilon} = \mathsf{LIT}_{\mathsf{b}}(\{\!|E|\!\}_{\mathsf{v}\epsilon}), \{\!|\underline{d}|\!\}_{\mathsf{v}\epsilon} = \mathsf{CST}(\wr d \wr), \{\!|\underline{\lambda} x.E|\!\}_{\mathsf{v}\epsilon} = \underline{\lambda}^{\mathsf{v}\epsilon} \lambda x.\{\!|E|\!\}_{\mathsf{v}\epsilon},$

$\{\!|E_1 \underline{@} E_2|\!\}_{\mathsf{v}\epsilon} = \mathsf{APP}(\{\!|E_1|\!\}_{\mathsf{v}\epsilon}, \{\!|E_2|\!\}_{\mathsf{v}\epsilon}), \{\!|\#E|\!\}_{\mathsf{v}\epsilon} = \underline{\#}^{\mathsf{v}\epsilon}(\{\!|E|\!\}_{\mathsf{v}\epsilon}),$

$\{\!|\underline{\mathbf{let}}\ x \Leftarrow E_1\ \underline{\mathbf{in}}\ E_2|\!\}_{\mathsf{v}\epsilon} = \underline{\mathsf{let}}^{\mathsf{v}\epsilon} \{\!|E_1|\!\}_{\mathsf{v}\epsilon} \lambda x.\{\!|E_2|\!\}_{\mathsf{v}\epsilon}, \{\!|\langle \mathcal{O} \rangle|\!\}_{\mathsf{v}\epsilon} = \{\!|\mathcal{O}|\!\}_{\langle\rangle\epsilon}$

"Evaluated" terms:

$\{\!|\$_{\mathsf{b}} \ell|\!\}_{\langle\rangle\epsilon} = \mathsf{LIT}_{\mathsf{b}}(\ell), \{\!|v_i|\!\}_{\langle\rangle\epsilon} = \mathsf{VAR}(i), \{\!|\underline{\lambda} v_i.\mathcal{O}|\!\}_{\langle\rangle\epsilon} = \mathsf{LAM}(i, \{\!|\mathcal{O}|\!\}_{\langle\rangle\epsilon}),$

$\{\!|\mathcal{O}_1 \underline{@} \mathcal{O}_2|\!\}_{\langle\rangle\epsilon} = \mathsf{APP}(\{\!|\mathcal{O}_1|\!\}_{\langle\rangle\epsilon}, \{\!|\mathcal{O}_2|\!\}_{\langle\rangle\epsilon}), \{\!|\underline{d}|\!\}_{\langle\rangle\epsilon} = \mathsf{CST}(\wr d \wr),$

$\{\!|\underline{\mathbf{let}}\ v_i \Leftarrow \mathcal{O}_1\ \underline{\mathbf{in}}\ \mathcal{O}_2|\!\}_{\langle\rangle\epsilon} = \mathsf{LET}(i, \{\!|\mathcal{O}_1|\!\}_{\langle\rangle\epsilon}, \{\!|\mathcal{O}_2|\!\}_{\langle\rangle\epsilon})$

Bindings:

$\{\!|\cdot|\!\}_{\mathsf{v}\epsilon} = \mathsf{nil}$

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

where we use the following terms:

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

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

Note also that α-conversion is not preserved by the embedding translation: bound variables v_i are translated to integer i under the translation for the "eval-

uated" terms. As a consequence, a general substitution lemma of the following form fails for this translation.

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

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

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

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

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

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

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

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

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

Lemma C.22 (Substitution lemma for well-formed terms). *If $\lambda x.E$ is well-formed, then $\{\!|E\{E'/x\}|\!\}_{\mathsf{v}\epsilon} \equiv \{\!|E|\!\}_{\mathsf{v}\epsilon}\{\{\!|E'|\!\}_{\mathsf{v}\epsilon}/x\}$.*[2]

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

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

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

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

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

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

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

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

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

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

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

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

We perform a case analysis on E.

Case $E \equiv \ell$**:** Simple.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

a. $\Delta \rhd [B_2]E'\{\mathbf{fix}\,\lambda x.E'/x\} \Downarrow [B_3]V$, $\{\!|V|\!\}^C_{\mathsf{v}\epsilon} = iV$; and

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

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

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

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

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

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

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

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

Note that

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

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

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

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

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

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

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

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

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

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

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

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

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

□

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

We can then construct derivations for

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

The conclusion follows immediately.

Case [*let*]**:** Similar to the case of rule ([*lam*]).

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

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

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

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

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

C.4 Call-by-value type-directed partial evaluation

C.4.1 Semantic correctness

Lemma C.27. *For all types* σ, $\mathsf{nPCF} \vdash \ \triangleright\, |{\downarrow^\sigma}| = \lambda x.x : \sigma \to \sigma$ *and* $\mathsf{nPCF} \vdash \triangleright\, |{\uparrow_\sigma}| = \lambda x.x : \sigma \to \sigma$.

Proof. By a straightforward induction on type τ. □

Theorem C.28 (Semantic correctness of TDPE). *If* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \triangleright E : \sigma^{\mathfrak{d}}$ *and* $\mathsf{vPCF}^{2} \vdash \mathsf{NF}(E) \Downarrow \mathcal{O}$, *then* $\mathsf{vPCF} \vdash \triangleright\, |\mathcal{O}| = |E| : \sigma$.

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

Proof. Similar to the proof of the corresponding theorem in the call-by-name case, Theorem 4.12, but using Lemma C.27 and Theorem 5.6 instead. □

C.4.2 Syntactic correctness

Theorem 5.8 (Refined type preservation). *If* $\mathsf{vPCF}^{2} \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B]E : \tau$ *and* $\mathsf{vPCF}^{2} \vdash \Delta \rhd [B]E \Downarrow [B']V$*, then* $\mathsf{vPCF}^{2} \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright [B']V : \tau$*.*

Proof. (Sketch) Similar to the proof of Theorem 5.2. As always, the most non-trivial case is the rule ([*app*]), for which we prove a substitution lemma for the refined type system (similar to Lemma B.9.) □

Corollary 5.9 (Refined type preservation for complete programs). *If* $\blacktriangleright E : \textcircled{e}^{nc}(\sigma)$ *and* $E \searrow \mathcal{O}$*, then* $\blacktriangleright \mathcal{O} : \textcircled{e}^{nc}(\sigma)$*.*

Theorem 5.10 (Normal-form code types). *If* V *is an* vPCF^{2}*-value (Figure 5.1), and* $\mathsf{vPCF}^{2} \vdash \textcircled{v}^{var}(\Delta) \blacktriangleright V : \textcircled{v}^{X}(\sigma)$ *where* X *is av, nv, bd, or nc, then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\Delta \rhd^{X} |\mathcal{O}| : \sigma$*.*

Proof. Similar to the proof of Theorem 4.14. □

Lemma 5.11. *(1) The extraction functions (Figure 5.3c) have the following normal-form types (writing* $\sigma^{\bigcirc \mathsf{nv}}$ *for* $\sigma\{\textcircled{v}^{nv}(\mathsf{b})/\mathsf{b} : \mathsf{b} \in \mathbb{B}\}$*.)*

$$\blacktriangleright \Downarrow^{\sigma} : \sigma^{\bigcirc \mathsf{nv}} \to \textcircled{v}^{nv}(\sigma), \blacktriangleright \Uparrow_{\sigma} : \textcircled{v}^{av}(\sigma) \to \sigma^{\bigcirc \mathsf{nv}}.$$

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

Proof.

(1) By induction on type σ.

Case $\sigma = \mathsf{b}$: Because at the base type, $\mathsf{b}^{\bigcirc\mathsf{nv}} = \textcircled{v}^{nv}(\mathsf{b})$, we just need to show: $\blacktriangleright\ \lambda x.x : \textcircled{v}^{nv}(\mathsf{b}) \to \textcircled{v}^{nv}(\mathsf{b})$ for $\downarrow^{\mathsf{b}}$, and $\blacktriangleright\ \lambda x.x : \textcircled{v}^{av}(\mathsf{b}) \to \textcircled{v}^{nv}(\mathsf{b})$ for $\uparrow_{\mathsf{b}}$. This is simple.

Case $\sigma = \sigma_1 \to \sigma_2$: Noting that $(\sigma_1 \to \sigma_2)^{\bigcirc\mathsf{nv}} = \sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}$, we give the following typing derivations, in the compact style used in the proof of Lemma 4.15.

- For $\blacktriangleright \downarrow^{\sigma_1\to\sigma_2} : (\sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}) \to \textcircled{v}^{nv}(\sigma_1 \to \sigma_2)$:

$$f : \sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}}, x : \textcircled{v}^{var}(\sigma_1) \blacktriangleright \underbrace{\downarrow^{\sigma_2}(\overbrace{f\,(\underbrace{\uparrow_{\sigma_1} x}_{\sigma_1^{\bigcirc\mathsf{nv}}})}^{\sigma_2^{\bigcirc\mathsf{nv}}})}_{\textcircled{v}^{nv}(\sigma_2)} : \textcircled{e}^{nc}(\sigma_2)$$

  Note the use of implicit coercions for term x and for $\downarrow^{\sigma_2}(f\,(\uparrow_{\sigma_1} x))$.

- For $\blacktriangleright \uparrow_{\sigma_1\to\sigma_2} : \textcircled{v}^{av}(\sigma_1 \to \sigma_2) \to (\sigma_1^{\bigcirc\mathsf{nv}} \to \sigma_2^{\bigcirc\mathsf{nv}})$:

$$e : \textcircled{v}^{av}(\sigma_1 \to \sigma_2), x : \sigma_1^{\bigcirc\mathsf{nv}} \blacktriangleright \underbrace{\uparrow_{\sigma_2}(\#(\overbrace{e\underline{@}(\underbrace{\downarrow^{\sigma_1} x}_{\textcircled{v}^{nv}(\sigma_1)})}^{\textcircled{e}^{bd}(\sigma_2)}))}_{\textcircled{v}^{var}(\sigma_2)} : \sigma_2^{\bigcirc\mathsf{nv}}$$

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

Theorem 5.12. *If* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \rhd E : \sigma^{\mathfrak{d}}$, *then* $\mathsf{vPCF}^{\imath} \vdash \blacktriangleright\ \mathsf{NF}(E) : \textcircled{v}^{nv}(\sigma)$.

Proof. By Lemma 5.11(2), we have $\mathsf{vPCF}^{\imath} \vdash \blacktriangleright\ \{\!|E|\!\}_{\mathsf{ri}} : \{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nv}}_{\mathsf{ri}}$. Since $\{\!|\sigma^{\mathfrak{d}}|\!\}^{\mathsf{nv}}_{\mathsf{ri}} \equiv \sigma^{\bigcirc\mathsf{nv}}$, applying $\downarrow^{\sigma} : \sigma^{\bigcirc\mathsf{nv}} \to \textcircled{v}^{nv}(\sigma)$ (Lemma 5.11(1)) to $\{\!|E|\!\}_{\mathsf{ri}}$ yields the conclusion. □

Corollary C.29 (Syntactic correctness of TDPE). *For* $\mathsf{vPCF}^{\mathsf{tdpe}} \vdash \triangleright E : \sigma^{\mathfrak{d}}$, *if* $\mathsf{vPCF}^{2} \vdash \mathsf{NF}(E) \searrow V$, *then* $V \equiv \mathcal{O}$ *for some* $\mathcal{O}$ *and* $\mathsf{vPCF} \vdash \Delta \triangleright^{nc} |\mathcal{O}| : \sigma$.

Proof. We use Theorem 5.12, Corollary 5.9, and Theorem 5.10. □

Appendix D

Notation and symbols

Meta-variables and fonts

E	(one-level or two-level) terms	
x, y, z	variables	
τ (resp. σ)	two-level (resp. one-level) types	35,36, 58,242
φ	two-level types in TDPE languages	47,65
θ	substitution-safe types (vPCF^2)	58
Γ (resp. Δ)	two-level/one-level typing contexts	35,36
B	accumulated bindings	64
b	base types	34
ℓ	literals (constants of base types)	34
d	dynamic constants in signature Sg	35
V	values (canonical terms)	35,64

$\mathcal{O}$	code-typed values	35,64
S	state	251
Sans serif ($\mathsf{bool}, \mathsf{CST}$)	syntax	
Underlined ($\underline{\lambda}x., \underline{@}$)	dynamic constructs	35,58
$d^{\mathfrak{d}}, \mathsf{b}^{\mathfrak{d}}$	dynamic constants and base types in TDPE languages	47,65

Language and Judgments

$\mathsf{L} \vdash J$	judgment J holds in language L.

General judgments

$\Gamma \rhd E : \tau$	term-in-context: "term E is of type τ under context Γ"
$\Gamma \rhd E_1 = E_2 : \tau$	equation-in-context: "E_1 and E_2 are equal terms of type τ under context Γ"

nPCF^2: call-by-name two-level language 35

$E \Downarrow V$	evaluation of a statically closed term	35
$\Gamma \blacktriangleright E : \tau$	term-in-context with refined typing for $\beta\eta$-normal object terms	52

$E\{\theta\}$	application of the substitution θ to E	

TDPE-specific notations

$\downarrow^{\sigma}$	reification function at type σ	47,65
$\uparrow_{\sigma}$	reflection function at type σ	47,65
$\{\!\{-\}\!\}_{\mathsf{ri}}$	residualizing instantiation	47,65
$\mathsf{NF}(-)$	static normalization function	47,65

Appendix E

Compiler generation for Tiny

E.1 A binding-time-separated interpreter for Tiny

Paulson's Tiny language [84] is a prototypical imperative language—the BNF of its syntax is given in Figure E.1. Figure E.2 displays the factorial function coded in Tiny.

Experiments in type-directed partial evaluation of a Tiny interpreter with respect to a Tiny program [11, 12] used an ML implementation of a Tiny interpreter (Figure E.3, page 274): For every syntactic category a meaning function is defined—see Figure E.4 (page 275) for the ML data type representing Tiny syntax. The meaning of a Tiny program is a function from store to store; the interpreter takes a Tiny program together with a initial store and, provided it terminates on the given program, returns a final store. Compilation by partially evaluating the interpreter with respect to a program thus results in the ML code of the store-to-store function denoted by the program.

```
program     ::=  block declaration in command end

declaration ::=  identifier*

command     ::=  skip
              |  command ; command
              |  identifier := expression
              |  if expression then command else command
              |  while expression do command end

expression  ::=  literal
              |  identifier
              |  (expression primop expression)

identifier  ::=  a string

literal     ::=  an integer

primop      ::=  + | - | * | < | =
```

Figure E.1: BNF of Tiny programs

```
block res val aux in
  aux := 1;
  while (0 < val) do
    aux := (aux * val);
    val := (val - 1)
  end;
  res := aux
end
```

Figure E.2: Factorial function in Tiny

Performing a binding-time analysis on the interpreter (under the assumptions that the input program is static and the input store is dynamic) classifies all the constants in the bodies of the meaning functions as dynamic; literals have to be lifted. As described at the end of Section 15.1, the interpreter is implemented within a functor that abstracts over all dynamic constants (for example `cond`, `fix`

```
fun meaning p store =
  let fun mp (PROGRAM (vs, c)) s                                  (* program *)
        = md vs 0 (fn env => mc c env s)
      and md [] offset k                                          (* declaration *)
        = k (fn i => ~1)
        | md (v :: vs) offset k
        = (md vs (offset + 1)
           (fn env => k (fn i => if v = i
                                 then offset
                                else env i)))
      and mc (SKIP) env s                                         (* command *)
        = s
        | mc (SEQUENCE(c1, c2)) env s
        = mc c2 env (mc c1 env s)
        | mc (ASSIGN(i, e)) env s
        = update (lift_int (env i), me e env s, s)
        | mc (CONDITIONAL(e, c_then, c_else)) env s
        = cond (me e env s,
                mc c_then env,
                mc c_else env,
                s)
        | mc (WHILE(e, c)) env s
        = fix (fn w => fn s
               => cond (me e env s,
                        fn s => w (mc c env s),
                        fn s => s,
                        s) ) s
      and me (LITERAL l) env s                                    (* expression *)
        = lift_int l
        | me (IDENTIFIER i) env s
        = fetch (lift_int (env i), s)
        | me (PRIMOP2(rator, e1, e2)) env s
        = mo2 rator (me e1 env s) (me e2 env s)
      and mo2 b v1 v2                                               (* primop *)
        =
        case b of
            Bop_PLUS => add (v1, v2)
          | Bop_MINUS => sub (v1, v2)
          | Bop_TIMES => mul (v1, v2)
          | Bop_LESS => lt (v1, v2)
          | Bop_EQUAL => eqi (v1, v2)
  in
    mp p store
  end
```

Figure E.3: An interpreter for Tiny

```
type Identifier = string

datatype
  Program =                                          (* program and declaration *)
    PROGRAM of Identifier list * Command
and
  Command =                                                        (* command *)
    SKIP                                                             (* skip *)
  | SEQUENCE of Command * Command                                      (* ; *)
  | ASSIGN of Identifier * Expression                                  (* := *)
  | CONDITIONAL of Expression * Command * Command                      (* if *)
  | WHILE of Expression * Command                                   (* while *)
and
  Expression =                                                   (* expression *)
    LITERAL of int                                                   (* literal *)
  | IDENTIFIER of Identifier                                      (* identifier *)
  | PRIMOP2 of Bop * Expression * Expression                        (* primop *)
and
  Bop =                                                            (* primop *)
    Bop_PLUS                                                           (* + *)
  | Bop_MINUS                                                          (* - *)
  | Bop_TIMES                                                          (* * *)
  | Bop_LESS                                                           (* < *)
  | Bop_EQUAL                                                          (* = *)
```

Figure E.4: Datatype for representing Tiny programs

and `update` in `mc`). This allows one to easily switch between the evaluating instantiation |`meaning`| and the residualizing instantiation ⌜`meaning`⌝. For the evaluating instantiation we simply instantiate the functor with the actual constructs, for example

```
fun cond (b, kt, kf, s) = if b <> 0 then kt s else kf s


fun fix f x             = f (fix f) x
```

For the residualizing instantiation ⌜`meaning`⌝ we instantiate the dynamic constants with code-generation functions; as pointed out in Example 2.4 (page 20) and

made precise in Definition 15.4 (page 157), reflection can be used to write code-generation functions:

```
fun cond e  = reflect (rrT4 (a', a' -!> a', a' -!> a', a')
                       -!> a')
                      (VAR "cond") e
fun fix f x = reflect (((a' -!> a') --> (a' -!> a')) -->
                       (a' -!> a'))
                      (VAR "fix") f x
```

E.2 Generating a compiler for Tiny

As mentioned in Chapter 17.6, we derive a compiler for Tiny in three steps:

1. rewrite `tiny_pe` into a functor `tiny_ge(S:STATIC D:DYNAMIC)` such that `meaning` is also parameterized over all static constants and base types
2. give instantiations of `S` and `D` as indicated by the instantiation table in Table 17.1 (page 184), thereby creating the GE-instantiation `"meaning"`
3. use the GE-instantiation `"meaning"` to perform the second Futamura projection

The following two sections describe the first two steps in more detail. Once we have a GE-instantiation, the third step is easily carried out with the help of an interface similar to the one for visualization described in Section 17.4.

E.3 "Full parameterization"

Following Section 17.3 we re-implement the interpreter inside a functor to parameterize over *both* static and dynamic base types and constants. Note, however, that the original implementation of Figure E.3 (page 274) makes use of recursive definitions and case distinctions; both constructs cannot be parameterized over directly. Hence we have to express recursive definitions with a fixed point operator and case distinctions with appropriate elimination functions. Consider for example case distinction over `Expression`; Figure E.5 shows the type of the corresponding elimination function.

```
val case_Expression
  : Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a
```

Figure E.5: An elimination function for expressions

The resulting implementation is sketched in Figure E.6. The recursive definition is handled by a top-level fixed point operator, and all the case distinctions have been replaced with a call to the corresponding elimination function.

Now that we are able to parameterize over every construct, we enclose the implementation in a functor as shown in Figure E.7 (page 279). The functor takes two structures; their respective signatures `STATIC` and `DYNAMIC` declare names for all base types and constants that are used statically and dynamically, respectively. A base type (for example `int`) may occur both statically (`int`) and

```
fun meaning p store =
  let val (mp, _, _, _, _) =
    fix5
    (fn (mp, md, mc, me, mo2) =>
     let fun mp' prog                                              (* program *)
           = ...
         and md' idList                                          (* declaration *)
           = ...
         and mc' c                                                (* command *)
           = (case_Command c
              (                                          (* mc (SKIP) env s *)
               fn _ => fn env => fn s
               => s,
                                                 (* mc (SEQUENCE(c1, c2)) env s *)
               fn (c1, c2) => fn env => fn s
               => mc c2 env (mc c1 env s),
                                                    (* mc (ASSIGN(i, e)) env s *)
               fn (i, e) => fn env => fn s
               => update (lift_int (env i), me e env s, s),
                                 (* mc (CONDITIONAL(e,c_then,c_else)) env s *)
               fn (e, c_then, c_else) => fn env => fn s
               => cond (me e env s,
                        mc c_then env,
                        mc c_else env,
                        s),
                                                    (* mc (WHILE (e, c)) env s *)
               fn (e, c) => fn env => fn s
               => fix (fn w
                       => fn s
                       => cond (me e env s,
                                fn s => w (mc c env s),
                                fn s => s,
                                s)) s
               ))
         and me' e                                               (* expression *)
           = (case_Expression e (...))
         and mo2' bop                                              (* primop *)
           = (case_Bop bop (...))
     in
       (mp', md', mc', me', mo2')
     end)
  in
    mp p store
  end
```

Figure E.6: A fully parameterizable implementation

```
functor tiny_ge (structure S : STATIC
                 structure D : DYNAMIC
                   sharing type S.Int_s = D.Int_s
                           ⋮ )=
  struct
    local open S D
    in
      fun meaning p store
        = ...
    end
  end
```

Figure E.7: Parameterizing over both static and dynamic constructs

dynamically (`int`$^{\mathfrak{d}}$)—in this case two distinct names (for example `Int_s` and `Int_d`) have to be used.

As mentioned in Chapter 17.6, the monotype of every instance of a constant appearing in the interpreter has to be determined. It is these monotypes that must be declared in the signatures `STATIC` and `DYNAMIC`. Figure E.8 shows a portion of signature `STATIC`: The polymorphic type of `caseExpression` (Figure E.5, page 277) gives rise to a type abbreviation `case_Exp_type`, which can be used to specify the types of the different instances of `caseExpression`. Note that if a static polymorphic constant is instantiated with a type that contains dynamic base types—like `Int_d` in the case of `caseExpression`—then these dynamic base types have to be included in the signature `STATIC` of static constructs.[1] For base

[1] Note that static base types appear also in the signature of dynamic constructs, because we make the lifting functions part of the latter. However there is a conceptual difference: in a two-level language, it is natural that the dynamic signature has dependencies on the static signature, whereas the static signature should not depend on the dynamic signature.

types that occur both in signatures `STATIC` and `DYNAMIC`, sharing constraints have to be declared in the interface of functor `tiny_ge` (Figure E.7).

```
⋮
type 'a case_Exp_type                                    (* Type abbreviation *)
  = Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a

type case_Exp_res_type                                   (* Result type *)
  = (Identifier -> Int_s) -> sto -> Int_d

⋮
                              (* Declaration of elimination function for expressions *)
val case_Expression: case_Exp_res_type case_Exp_type

⋮
```

Figure E.8: Excerpts from signature `STATIC`

Finding the monotypes for the various instantiations of constants in the interpreter can be facilitated by using the type-inference mechanism of ML: We transcribe the output of ML type inference into a type specification by hand. This transcription is straightforward, because the type specifications of TDPE and the output of ML type inference are very much alike.

E.4 The GE-instantiation

After parameterizing the interpreter as described above, we are in a position to either run the interpreter by using its evaluating instantiation (see Definition 15.5 (page 158)), perform type-directed partial evaluation by employing

the residualizing instantiation (Definition 15.4 (page 157)), or carry out the second Futamura projection with the GE-instantiation (Definition 16.6 (page 176)). Section 17.3 shows how the static and dynamic constructs have to be instantiated in each case. For the GE-instantiation, all base types become `Exp`; static and dynamic constants are instantiated with code-generation functions. The latter are constructed using the evaluating and the residualizing instantiation of reflection, respectively. Because the signatures `STATIC` and `DYNAMIC` hold the precise type at which each constant is used, it is purely mechanical to write down the structures needed for the GE-instantiation.

Bibliography

[1] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268., April 1991.

[2] Martin Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] Alex Aiken, editor. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio, Texas, January 1999.

[4] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[5] Alan Bawden. Quasiquotation in Lisp. In Danvy [16], pages 4–12. Available online at `http://www.brics.dk/~pepm99/programme.html`.

[6] 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.

[7] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 198–214, Madrid, Spain, September 1994.

[8] Hans-J. Boehm, editor. *Proceedings of the Twenty-First Annual ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 1994.

[9] 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.

[10] Cristiano Calcago, Eugenio Moggi, and Walid Taha. Closed types as a simple approach to safe imperative multi-stage programming. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *27th International Colloquium on Automata, Languages, and Programming*, volume 1853 of *Lecture Notes in Computer Science*, pages 25–36, Geneva, Switzerland, July 2000.

[11] Olivier Danvy. Pragmatics of type-directed partial evaluation. In Danvy et al. [19], pages 73–94. Extended version available as the technical report BRICS RS-96-15.

[12] Olivier Danvy. Type-directed partial evaluation. In Steele [94], pages 242–257.

[13] Olivier Danvy. Functional unparsing. Technical Report BRICS RS-98-12, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1998. Supersedes the earlier report BRICS RS-98-5. Extended version of an article to appear in the Journal of Functional Programming.

[14] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages, and Programming*, number 1443 in Lecture Notes in Computer Science, pages 908–917. Springer-Verlag, 1998.

[15] 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.

[16] Olivier Danvy, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Antonio, Texas, January 1999. ACM Press.

[17] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM*

Workshop on Normalization by Evaluation, NBE '98, (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Note Series, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[18] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[19] Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996.

[20] Olivier Danvy and John Hatcliff. CPS transformation after strictness analysis. *ACM Letters on Programming Languages and Systems*, 1(3):195–212, 1993.

[21] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[22] Olivier Danvy and Morten Rhiger. Compiling actions by partial evaluation, revisited. Technical Report BRICS RS-98-13, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1998.

[23] 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.

[24] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Extended version available as BRICS technical report RS-96-13.

[25] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In Swierstra [98], pages 224–242.

[26] 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.

[27] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Steele [94], 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.

[28] Peter Harry Eidorff, Fritz Henglein, Christian Mossin, Henning Niss, and Morten Heine Sørensen. AnnoDomini: From type theory to Year 2000 conversion tool. In Aiken [3], pages 1–14.

[29] Andrzej Filinski. Representing monads. In Boehm [8], pages 446–457.

[30] Andrzej Filinski. Representing layered monads. In Aiken [3], pages 175–188.

[31] 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.

[32] 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.

[33] 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.

[34] The FoxNet homepage. `http://foxnet.cs.cmu.edu`.

[35] 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.

[36] Jean-Yves Girard. The system F of variable types, fifteen years later. *Theoretical Computer Science*, 45(2):159–192, 1986.

[37] 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.

[38] Bernd Grobauer and Zhe Yang. The second Futamura projection for type-directed partial evaluation. In Lawall [66], pages 22–32. Extended version to appear in the journal *Higher-Order and Symbolic Computation*, 14(2/3), 2001.

[39] Bernd Grobauer and Zhe Yang. Source code for the second Futamura projection for type-directed partial evaluation in ML, 2000. Available from `http://www.brics.dk/~tdpe/second_FP/sources.tgz`.

[40] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. The MIT Press, Cambridge, Massachusetts, 1992.

[41] Cordelia Hall, Kevin Hammond, Simon Peyton-Jones, and Philip Wadler. Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, 18(2):109–138, March 1996.

[42] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In Peter Lee, editor, *Proceedings of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, California, January 1995.

[43] John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In Boehm [8], pages 458–471.

[44] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–541, 1997.

[45] Fritz Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 22(3):197–230, June 1994.

[46] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In Simon Peyton Jones, editor, *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 192–203, La Jolla, California, June 1995.

[47] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[48] Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In Rogardt Heldal, Cartsen K. Holst, and Philip L. Wadler, editors, *Draft Proceedings, 4^{th} Annual Glasgow Workshop on Functional Programming*, Workshops in Computing, pages 210–218, Skye, Scotland, 1991. Springer-Verlag.

[49] John Hughes. The design of a pretty-printing library. In Jansson Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 53–96. Springer-Verlag, 1995.

[50] Patrik Jansson and Johan Jeuring. PolyP—a polytypic programming language extension. In Jones [56], pages 470–482.

[51] John McCarthy *et al. LISP 1.5 Programmer's Manual.* The MIT Press, Cambridge, Massachusetts, 1962.

[52] Mark P. Jones. A system of constructor classes: overloading and implicit higher-order polymorphism. *Journal of Functional Programming*, 5(1):1–35, January 1995. An earlier version appeared in FPCA '93.

[53] Mark P. Jones. First-class polymorphism with type inference. In Jones [56], pages 483–496.

[54] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.

[55] Neil D. Jones, editor. *Special issue on Partial Evaluation*, Journal of Functional Programming, Vol. 3, Part 3. Cambridge University Press, July 1993.

[56] Neil D. Jones, editor. *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.

[57] 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`.

[58] Neil D. Jones and Steven S. Muchnick. *TEMPO: A Unified Treatment of Binding Time and Parameter Passing Concepts in Programming Lan-*

guages, volume 66 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[59] 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.

[60] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998. Available online at `http://www.brics.dk/~hosc/11-1/`.

[61] Andrew Kennedy. Relational parametricity and units of measure. In Jones [56], pages 442–455.

[62] 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.

[63] 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.

[64] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.

[65] Peter J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[66] 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.

[67] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, pages 227–238, Orlando, Florida, June 1994.

[68] Peter Lee. *Realistic Compiler Generation*. The MIT Press, 1989.

[69] 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.

[70] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[71] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[72] John C. Mitchell. Coercion and type inference. In Ken Kennedy, editor, *Proceedings of the Eleventh Annual ACM Symposium on Principles*

of Programming Languages, pages 175–185, Salt Lake City, Utah, January 1984.

[73] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[74] 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.

[75] 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.

[76] Eugenio Moggi, Walid Taha, Zine El-Abidine Benaissa, and Tim Sheard. An Idealized MetaML: Simpler, and more expressive. In Swierstra [98], pages 193–207.

[77] Lockwood Morris. The next 700 formal language descriptions. *Lisp and Symbolic Computation*, 6(3/4):249–258, 1993.

[78] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1992.

[79] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[80] Flemming Nielson and Hanne Riis Nielson. Multi-level lambda-calculi: an algebraic description. In Danvy et al. [19], pages 338–354.

[81] Flemming Nielson and Hanne Riis Nielson. Prescriptive frameworks for multi-level lambda-calculi. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 193–202, Amsterdam, The Netherlands, June 1997.

[82] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In Steele [94], pages 54–67.

[83] Jens Palsberg. Correctness of binding-time analysis. In Jones [55], pages 347–363.

[84] Lawrence C. Paulson. Compiler generation from denotational semantics. In Bernard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.

[85] John Peterson, Kevin Hammond, et al. Report on the programming language Haskell, a non-strict purely-functional programming language, version 1.4. Available at the Haskell homepage: http://www.haskell.org, April 1997.

[86] 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.

[87] Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[88] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, December 1977.

[89] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report FN-19, Department of Computer Science, University of Aarhus, Aarhus, Denmark, September 1981.

[90] 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.

[91] Calton Pu, Henry Massalin, and John Ioannidis. The Synthesis kernel. *Computing Systems*, 1(1):11–32, winter 1988. University of California Press.

[92] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, number 19 in Lecture Notes in Computer Science, pages 408–425, Paris, France, April 1974.

[93] Mark Shields, Tim Sheard, and Simon Peyton Jones. Dynamic typing as staged type inference. In Luca Cardelli, editor, *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 289–302, San Diego, California, January 1998.

[94] Guy L. Steele, editor. *Proceedings of the Twenty-Third Annual ACM Sym-*

posium on Principles of Programming Languages, St. Petersburg Beach, Florida, January 1996.

[95] Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[96] Eijiro Sumii, 2000. Email exchange, February 2000.

[97] Eijiro Sumii and Naoki Kobayashi. Online-and-offline partial evaluation: A mixed approah. In Lawall [66], 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.

[98] S. Doaitse Swierstra, editor. *Proceedings of the Eighth European Symposium on Programming*, Amsterdam, Netherlands, March 1999.

[99] Walid Taha. A sound reduction semantics for untyped CBN multi-stage computation. or, the theory of MetaML is non-trival (extended abstract). In Lawall [66], pages 34–43.

[100] Walid Taha and Tim Sheard. Multi-stage programming. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 321–321, Amsterdam, The Netherlands, June 1997.

[101] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[102] Mads Tofte. Principal signatures for higher-order program modules. *Journal of Functional Programming*, 4(3):285–335, July 1994.

[103] René Vestergaard. From proof normalization to compiler generation and type-directed change-of-representation. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, May 1997.

[104] Mitchell Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496–517, 1982.

[105] Mitchell Wand. Specifying the correctness of binding-time analysis. In Jones [55], pages 365–387.

[106] 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.

[107] Glynn Winskel. *The Formal Semantics of Programming Languages*. Foundation of Computing Series. The MIT Press, 1993.