Basic Research in Computer Science

BRICS RS-99-40 Grobauer & Yang: The Second Futamura Projection for Type-Directed Partial Evaluation

The Second Futamura Projection for Type-Directed Partial Evaluation

Bernd Grobauer
Zhe Yang

BRICS Report Series **RS-99-40**

ISSN 0909-0878 **November 1999**

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

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

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/99/40/`

The Second Futamura Projection for Type-Directed Partial Evaluation*

Bernd Grobauer †

BRICS‡
Department of Computer Science
University of Aarhus

Zhe Yang §

Department of Computer Science
New York University

November 1999

Abstract

A generating extension of a program specializes it with respect to some specified part of the input. A generating extension of a program can be formed trivially by applying a partial evaluator to the program; the second Futamura projection describes the automatic generation of non-trivial generating extensions by applying a partial evaluator to itself with respect to the programs.

We derive an ML implementation of the second Futamura projection for Type-Directed Partial Evaluation (TDPE). Due to the differences between 'traditional', syntax-directed partial evaluation and TDPE, 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. In the context of the second Futamura projection, we also compare and relate TDPE with conventional offline partial evaluation.

We demonstrate our technique with several examples, including compiler generation for Tiny, a prototypical imperative language.

*A preliminary version of this paper appeared in the Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'00).

†Ny Munkegade, Building 540, 8000 Aarhus C, Denmark.
E-mail: grobauer@brics.dk

‡Basic Research in Computer Science (www.brics.dk),
Centre of the Danish National Research Foundation.

§251 Mercer Street, New York, NY 10012, USA.
E-mail: zheyang@cs.nyu.edu
Work done while visiting BRICS.

1 Introduction

1.1 Background

General Notions 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., $[\![p_s\, d]\!] = [\![p(s, d)]\!]$. Often, some computation in p can be carried out independently of the dynamic input d, and hence the specialized program p_s is more efficient than what 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., $[\![PE(\ulcorner p \urcorner, \ulcorner s \urcorner)]\!]$ (where $\ulcorner \cdot \urcorner$ denotes "the encoding of") is a specialized program p_s, if PE terminates on $\ulcorner p \urcorner$ and $\ulcorner s \urcorner$.

The program $\lambda s.PE(\ulcorner p \urcorner, s)$, which computes a specialized program p_s for any input s, is an instance of a *generating extension* of program p. In general, a program p' is a generating extension of the program p, if running p' on $\ulcorner s \urcorner$ yields a specialization of p with respect to the static input s (under the assumption that p' terminates on $\ulcorner s \urcorner$). Since PE itself is a general program, and a specific program p is often available before its particular static input s, it makes sense to specialize the program PE with respect to the program p to produce a more efficient generating extension PE_p. 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 follows:

$$p' = [\![PE(\ulcorner PE \urcorner, \ulcorner p \urcorner)]\!].$$

Self-application The above equation was first observed in 1971 by Futamura [14] in the context of compiler generation—the generating extension of an interpreter is a compiler—and is called the *second Futamura projection*. Turning this equation into practice, however, proved to be much more difficult than the simple equational form suggests; it was not until 1985 that Jones's group implemented Mix [20], 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 making the decision *offline*: pre-annotate the source program p with binding-time annotations which

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 (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 [2] and Schism [4], but most of them are for untyped languages. For typed languages, the so-called *type specialization* problem arises [18]: since a (traditional) partial evaluator PE must be able to perform all the static subcomputations, in particular it must subsume a traditional evaluator. Generally, such an evaluator uses a universal data type to represent values in the evaluated program. The resulting generating extension p', then, often retains the universal data type and the tagging/untagging operations as a source of overhead. Partly because of this, in the 1990's, the practice shifted towards hand-written generating-extension generators [17]; this is also known as the *cogen approach*. The relationship between a generating-extension generator and a corresponding partial evaluator is like the familiar relationship between a compiler and an interpreter. It takes more effort to write a generating-extension generator than to write just a partial evaluator. In the case of self application, this extra effort is taken care of by reusing the partial evaluator itself. Furthermore, apart from technology reuse, a related issue is correctness arguments: proving the correctness of a hand-written generating-extension generator is more difficult than proving the correctness of a partial evaluator; on the other hand, if a partial evaluator has been proved correct, the correctness of a generating extension produced by self-applying this partial evaluator follows immediately. Furthermore, as we shall see in this article, the problem caused by using 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.

Type-directed partial 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 $p : \tau_1 \to \tau_2$ and $s : \tau_1$, bringing the application ps into β-normal form specializes p with respect to s. For example specializing the K-combinator $K = \lambda\, x.\lambda\, y.x$ with respect to itself yields $\lambda\, y.\lambda\, x.\lambda\, y'.x$.

TDPE, due to Danvy [6], realizes the above idea using a technique called *Normalization by Evaluation* (NbE) of Berger and Schwichtenberg [1]. 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 1. *Let PL be a higher-order functional language in which we can define a type* `Exp` *of term representations. Consider the combinator* $K = \lambda x.\lambda y.x$*—the term* KK *is of type* `Exp`$\to$`Exp` $\to$ `Exp` $\to$ `Exp`*. We want to extract*

its long $\beta\eta$-normal form from its meaning.

Since $\texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$ *is the type of a function which takes three arguments, one can infer that the long $\beta\eta$-normal form of KK must be of the form $\lambda\, v_1.\lambda\, v_2.\lambda\, v_3.\texttt{t}$, for some term $\texttt{t} : \texttt{Exp}$. Intuitively, the only natural way to generate the term* `t` *from the meaning of term KK is to apply it to (the values representing) terms v_1, v_2 and v_3. The result of this application is the term v_2. Thus, we can extract the normal form of KK as $\lambda\, v_1.\lambda\, v_2.\lambda\, v_3.v_2$.*

TDPE is different from a traditional, syntax-directed partial evaluator in several respects:

Binding-Time Annotation Traditional partial evaluators require binding-time annotations for all the subexpressions. 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. As we can see from Example 1, function abstraction and application are always carried out as static computation; all β-redexes are eliminated this way. The remaining constructions are "coerced" into code representations by the extraction function.

Type-Directed The construction of the extraction function relies on the type of the term to be normalized, making TDPE "type-directed".

Efficiency Traditional partial evaluators work by symbolic computation on the source programs; they contain an evaluator to perform the static evaluation and code generation. TDPE reuses the underlying evaluator (interpreter or compiler) to perform these operations; given a highly optimized evaluation mechanism of functional languages, TDPE acquires the efficiency for free—a feature shared with the cogen approach.

Compared with a symbolic normalization technique, TDPE is faster also because there is no explicit rewriting steps; every term will be generated only once.

Flexibility Traditional partial evaluators need to be able to deal with all the constructs in the language. In TDPE, one is free to use various syntactic constructs, such as pattern matching, since only the meaning of the term, along with its type, determines the normal form, not how the term is written.

These differences have contributed to the successful application of TDPE in various contexts, e.g., to perform semantics-based compilation [11]. An introductory account, as well as a survey of various treatments concerning NbE, can be found in Danvy's lecture notes [8].

1.2 Our Work

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.

- Since TDPE extracts the normal form of a term according to its type, a natural self-application of TDPE, one which has been done by Danvy in his original article for the untyped language Scheme [6], is to specialize the TDPE process with respect to a particular type. The result helps one to visualize what a particular instance of TDPE does. But this form of self-application does not further specialize with respect to a particular source program. In addition, it is also not clear if self-application can be achieved in a typed setting, such as in ML, since the extraction functions are type-indexed. Indeed, the ML implementation of TDPE encodes a type as the application of several combinators that correspond to different type constructors; the TDPE algorithm to be specialized is not a monolithic program.

- Following the second Futamura projection literally, one should specialize the source program of the partial evaluator. Partial evaluation using TDPE is carried out using the underlying evaluator, whose source code might be written in an arbitrary language or not even available. 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 deals with monomorphically typed programs, but the standard call-by-value TDPE algorithm itself uses polymorphically typed control operators shift and reset to perform let-insertion in a polymorphically typed evaluation context.

Our contribution This article addresses all the above issues. We show how to effectively carry out self-application for TDPE, in a strongly typed language, ML. 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 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 object program involved in self-application, we provide a somewhat detailed introduction to the principle

and the implementation of TDPE in Section 2. Section 3 provides an abstract account of our approach to self-application for TDPE, and Section 4 details the development in the context of ML. Section 5 describes the derivation of the Tiny compiler. Based on our experiments, we give some benchmarks in Section 6. Section 7 concludes. The appendix gives further technical details in the generation of a Tiny compiler. The complete sources of the development presented in this article are available online [15].

2 TDPE in a nutshell

In order to give some intuition, we first outline TDPE for an effect-free fragment of ML without recursion (Section 2.1). Then we sketch the extensions and pragmatic issues of TDPE in a larger subset of ML, the setting we will work with in the later sections (Section 2.2). Finally, to facilitate a precise formulation of self-application, we outline Filinski's formalization of TDPE (Section 2.3).

2.1 Pure TDPE in ML

Pure TDPE deals with an effect-free fragment of ML without recursion, where the call-by-name and call-by-value semantics agree.

Pure simply-typed λ-terms We first consider only pure simply-typed λ-terms. We use the type `Exp` in Figure 1 to represent code (as it is used in Example 1). In the following we will write $\underline{v}$ for `Var` v, $\underline{\lambda}\,\underline{v}.t$ for `LAM (`v`,` t`)` and $s\,\underline{@}\,t$ for `APP (`s`,` t`)`; 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 1: A data type for representing terms

Let us for now only consider ML functions which 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. ML polymorphism allows us to instantiate '$\bullet$' with `Exp` when coding such a λ-term in ML. So every λ-term of type τ gives rise to an ML value of type $\overline{\tau} = \tau[\bullet := \texttt{Exp}]$; that is, a value representing either code (when $\overline{\tau} = \texttt{Exp}$), or a code-manipulation function (at higher types).

Figure 2 shows the TDPE algorithm: for every type τ, we define inductively a pair of functions $\downarrow^\tau : \overline{\tau} \to \texttt{Exp}$ (reification) and $\uparrow_\tau : \texttt{Exp} \to \overline{\tau}$ (reflection).

Let us revisit the normalization of KK from Example 1:

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

Figure 2: Reification and reflection

Example 2. *For the type* $\bullet \to \bullet \to \bullet \to \bullet$ *the equations given in Figure 2 define reification as*

$$\downarrow^{\bullet\to\bullet\to\bullet\to\bullet} e = \underline{\lambda\, x}.\underline{\lambda\, y}.\underline{\lambda\, z}.e\underline{x}\underline{y}\underline{z} \qquad (x, y, z \text{ are fresh}).$$

Applying $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$ *to* KK *yields* $\underline{\lambda\, x}.\underline{\lambda\, y}.\underline{\lambda\, z}.\underline{y}$.

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 fresh variable, since the types do not match: $\ulcorner \tau_1 \urcorner \neq \texttt{Exp}$. This is where the *reflection* function $\uparrow_\tau : \texttt{Exp} \to \ulcorner \tau \urcorner$ comes in; it converts a code representation into a code-generating function:

Example 3. *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 their subsequent application to* e*. It is used, e.g., in 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 $\lambda\, z.\texttt{mult}\ 3.14\ z$ of type $\texttt{real} \to \texttt{real} \to \texttt{real}$, where `mult` is a curried version of multiplication over reals. There is no way this function can be used as a code-manipulating function. The solution is to use a non-standard, code-generating instantiation for the base types `real` and constants `mult`. We instantiate type `real` as $\texttt{real}_\texttt{r} = \texttt{Exp}$, and constant `mult` as some term $\texttt{mult}_\texttt{r} : \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$. We also *lift* the constant 3.14 into `Exp` using a lifting-function $\mathit{lift}_{\texttt{real}} : \texttt{real} \to \texttt{Exp}$. (This requires

a straightforward extension of the data type `Exp` with an additional constructor `LIT_REAL`.) Reflection can be used to construct the code-generating instantiation $\texttt{mult}_\mathsf{r}$:

Example 4. *A code-generating instantiation* $\texttt{mult}_\mathsf{r}$:`Exp`→`Exp` → `Exp` *of* `mult`: `real` → `real` → `real` *is given by*

$$\texttt{mult}_\mathsf{r} \;=\; \uparrow_{\bullet\to\bullet\to\bullet} \underline{\textit{“mult”}} \;=\; \lambda x.\lambda y.\underline{\textit{“mult”}}\,\underline{@}\,x\,\underline{@}\,y.$$

Now, applying the reification function $\downarrow^{\bullet\to\bullet}$ *to the term*

$$\lambda z.(\texttt{mult}_\mathsf{r}\ (\textit{lift}_{\texttt{real}}\ 3.14)\ z)$$

evaluates to $\underline{\lambda x.}\underline{\textit{“mult”}}\,\underline{@}\,\underline{3.14}\,\underline{@}\,\underline{x}.$

Partial evaluation In the framework of TDPE, partially evaluating a (curried) program $p : \sigma_S \to \sigma_D \to \sigma_R$ to a static input $s : \sigma_S$ is carried out by normalizing the application ps. We can always take the code-generating instantiation for all the base types and 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 of s to p enables also computation involving constants, not only β-reductions. Partial evaluation, of course, should also carry out such computation. Not surprisingly, this is achieved by instantiating the constants to themselves (this is called the *evaluating instantiation*).

Example 5. *Consider the function*

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

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. Hence we leave* `sin` *as it is (evaluating instantiation), but lift its result into* `Exp` *and give* `mult` *the code-generation instantiation introduced in Example 4:*

$$\texttt{height}_\mathsf{r} = \lambda a.\lambda z.\texttt{mult}_\mathsf{r}\ (\textit{lift}_{\texttt{real}}(\texttt{sin}\ a))\ z$$

Now we can specialize `height` *with respect to* $\frac{\pi}{6}$ *by evaluating* $\downarrow^{\bullet\to\bullet}(\texttt{height}_\mathsf{r}\ \frac{\pi}{6})$, *which yields* $\underline{\lambda x.}\underline{\textit{“mult”}}\,\underline{@}\,\underline{0.5}\,\underline{@}\,\underline{x}$

In general, to perform TDPE for a term, one needs to decide for each constant occurrence, whether to use the evaluating instantiation or the code-generation instantiation, and to insert appropriate lifting functions 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 corresponds to a binding-time annotation phase, which turns the source term into a well-formed two-level term. This will be made precise in the framework of a two-level language (Section 2.3).

2.2 TDPE in ML: implementation and extensions

Implementation Type-indexed functions such as reification and reflection can be implemented in ML employing a technique first used by Filinski and Yang [7, 26]; see also Rhiger's derivation [22]. A combinator is defined for every type constructor $\mathcal{T}$ ($\bullet$ and $\rightarrow$ in the case of Pure NbE in Section 2). This combinator takes a *pair* of reification and reflection functions for every argument τ_i to the (n-ary) type constructor $\mathcal{T}$, and computes the reification-reflection pair for the constructed type $\mathcal{T}(\tau_1, \ldots, \tau_n)$. 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.

As Figure 3 shows, we implement NbE as a functor parameterized over three structures of respective signatures `EXP` (term representation), `GENSYM` (name generation for variables) and `CONTROL` (control operators, used in "extensions" below). The implementation for Pure NbE shown in Figure 4 is a direct transcription from the formulation in Section 2.1.

Example 6. *With an implementation* `pNbe: NBE` *of Pure NbE (acquired by applying the functor* `pureNbe`*), normalization of KK (see Examples 1 and 2) is carried out as follows:*

```
local  open pNbe; infixr 5 -->  in
  val K = (fn x => fn y => x)
  val KK_norm = reify (a' --> a' --> a' --> a') (K K)
end
```

Extensions We will use a much extended version of TDPE, referred to as Full TDPE in this article. Full TDPE not only deals with the function type constructor, but also with tuples and sums. Furthermore, a complication which we have disregarded so far is that ML is a call-by-value (cbv) language with computational effects. In such languages, the rule of β-reduction is not sound because it might discard or duplicate computations with effects.

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.

One approach to handling sum types and call-by-value languages is to implement the reflection function by manipulating the code-generation context. This has been achieved by using the control operators shift and reset [9, 12]. Section 4.4 gives a more detailed treatment of dealing with sum types and call-by-value languages in TDPE.

An implementation of Full TDPE is described in Danvy's lecture notes [8]. The relevance of Full TDPE in this article is that (1) it is the partial

```
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;

signature CTRL =                                   (* control operators *)
  sig
    type Exp
    val shift: (('a -> Exp) -> Exp) -> 'a
    val reset: (unit -> Exp) -> Exp
  end;

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

functor nbe(structure G: GENSYM
            structure E: EXP
            structure C: CTRL): NBE = ...
```

Figure 3: NbE in ML

evaluator that one would use for specializing realistic programs; and (2) in particular, it handles all features used in its own implementation (e.g., name-generation uses side effects to maintain a counter). Hence in principle

```
functor pureNbe(structure G: GENSYM
                structure E: EXP): 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
      = (G.init (); reif v)
    fun reflect (RR (reif, refl)) e
      = refl e
  end
```

Figure 4: Implementation of Pure NbE

self-application should be possible.

2.3 A general account of TDPE

This section describes Filinski's formalization of TDPE [13]. The formal result makes precise how TDPE is used, which is important in deriving the formulation of self-application later.

Preliminaries First we fix some standard notions. A simple functional language 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. (The types are possibly polymorphic; however, in our technical development, we will only work with monomorphic instances.) A set of typing rules generates, from the signature Σ, typing judgments of the form $\Gamma \vdash_\Sigma t : \sigma$, which reads "$t$ is a well-formed term of type σ under typing context Γ".

Types and terms are associated with domain-theoretical denotations by an interpretation. An interpretation $\mathcal{I}$ of signature Σ assigns domains to base type names, and elements of appropriate domains to literals and constants (and, in the setting of cbv-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; to keep the presentation simple, we write $[\![t]\!]^{\mathcal{I}}$ only for closed terms t, which denote an element in the domain $[\![\sigma]\!]^{\mathcal{I}}$.

The syntactic counterpart of the notion of an interpretation is that of an instantiation. An instantiation maps syntactic phrases in a language L to syntactic phrases in (possibly) another language L', specified as a substitution Φ from the base types in language L to L'-types, and constants $c : \tau$ to L'-terms of type $\tau\{\Phi\}$. It is obvious that an interpretation of a language L' and an instantiation of a language L in language L' together determine an interpretation of L.

Two-level language Filinski formalized TDPE using a notion of two-level languages (or, binding-time-separated languages). The signature Σ^2 of such a language is the disjoint union of a static signature $\Sigma^{\mathfrak{s}}$ (static base types $\mathsf{b}^{\mathfrak{s}}$ and static constants $c^{\mathfrak{s}}$, written with superscript $\mathfrak{s}$), a dynamic signature $\Sigma^{\mathfrak{d}}$ (dynamic base types $\mathsf{b}^{\mathfrak{d}}$ and dynamic constants $c^{\mathfrak{d}}$, written with superscript $\mathfrak{d}$), and lifting functions $\$_{\mathsf{b}}$ for base types. For simplicity, we assume all static base types $\mathsf{b}^{\mathfrak{s}}$ are persistent, i.e., each of them has a corresponding dynamic base type $\mathsf{b}^{\mathfrak{d}}$, and is equipped with a lifting function $\$_{\mathsf{b}} : \mathsf{b}^{\mathfrak{s}} \to \mathsf{b}^{\mathfrak{d}}$. The intuition is that a value of a persistent base type always has a unique external representation as a constant, which can appear in the generated code; we call such a constant a *literal*. For TDPE, Filinski used a technical notion of *fully dynamic types*: a fully dynamic type is a type constructed solely from dynamic base types. The meaning $[\![e]\!]^{\mathcal{I}^2}$ of a term e is fully determined by a static interpretation $\mathcal{I}^{\mathfrak{s}}$ of signature $\Sigma^{\mathfrak{s}}$, and a dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$ of signature $\Sigma^{\mathfrak{d}}$ and the lifting functions. A two-level language is different from a one-level language in that it is specified by a pair $(\Sigma^2, \mathcal{I}^{\mathfrak{s}})$ of its signature Σ^2 and *only* a fixed static interpretation $\mathcal{I}^{\mathfrak{s}}$. In other words, the meaning of terms is parameterized over the dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$.

A two-level language $PL^2 = (\Sigma^2, \mathcal{I}^{\mathfrak{s}})$ is usually associated with a one-level language $PL = (\Sigma^{PL}, \mathcal{I}^{PL})$. Without loss of generality, we let the languages PL and PL^2 be fixed, where (1) the dynamic signature $\Sigma^{\mathfrak{d}}$ of PL^2 duplicates Σ^{PL} (except for literals, which can be lifted from static literals) with all constructs superscripted by $\mathfrak{d}$, (2) the static signature $\Sigma^{\mathfrak{s}}$ of PL^2 comprises all the base types in PL and all the constants in PL that have no computational effects except possible divergence, with all constructs superscripted by $\mathfrak{s}$, and (3) the static interpretation $\mathcal{I}^{\mathfrak{s}}$ is the restriction of interpretation $\mathcal{I}^{PL}$ to $\Sigma^{\mathfrak{s}}$. For clarity, we use metavariables t, e, σ, and τ to range over Σ^{PL}-terms, Σ^2-terms, Σ^{PL}-types, and Σ^2-types, respectively.

We can induce an *evaluating dynamic interpretation* $\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}$ from $\mathcal{I}^{PL}$ by taking $[\![\mathsf{b}^{\mathfrak{d}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}}$, $[\![c^{\mathfrak{d}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![c]\!]^{\mathcal{I}^{PL}}$, and $[\![\$]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = \lambda x \in [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}}.x$. A closely related notion is the *evaluating instantiation* of Σ^2-phrases in Σ^{PL}:

Definition 7 (Evaluating Instantiation). *The* evaluating instantiation *of a* Σ^2*-term* $\vdash_{\Sigma^2} \mathsf{e}{:}\tau$ *in PL is* $\vdash_{\Sigma^{PL}} \widetilde{\mathsf{e}}{:}\widetilde{\tau}$*, given by* $\widetilde{\mathsf{e}} = \mathsf{e}\{\Phi_\sim\}$ *and* $\widetilde{\tau} = \tau\{\Phi_\sim\}$*, where instantiation* $\Phi_\sim$ *is a substitution of* Σ^2*-constructs (constants and base types) into* Σ^{PL}*-phrases (terms and types):* $\Phi_\sim(\mathsf{b}^\mathfrak{s}) = \Phi_\sim(\mathsf{b}^\mathfrak{d}) = \mathsf{b}$*,* $\Phi_\sim(c^\mathfrak{s}) = \Phi_\sim(c^\mathfrak{d}) = c$*,* $\Phi_\sim(\$_b) = \lambda x.x$*.*

We have that for all Σ^2*-types* τ *and* Σ^2*-terms* e*,* $[\![\widetilde{\tau}]\!]^{\mathcal{I}^{PL}} = [\![\tau]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}_{\mathsf{ev}}}$ *and* $[\![\widetilde{\mathsf{e}}]\!]^{\mathcal{I}^{PL}} = [\![\mathsf{e}]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}_{\mathsf{ev}}}$*.*

Static normalization A Σ^2-term of some fully dynamic type is in *static normal form* if it is free of β-redexes and free of static constants, except literals that appear as arguments to lifting functions; in other words, the term cannot be further simplified without knowing the interpretations of the dynamic constants. Terms e in static normal form are, in fact, in one-to-one correspondence to terms $\widetilde{\mathsf{e}}$ in Σ^{PL}. They can thus be represented using a one-level term representation such as the one provided by `Exp`.

A static normalization function *NF* for PL^2 is a computable partial function on well-typed Σ^2-terms such that if $\mathsf{e}' = NF(\mathsf{e})$ then e' is a Σ^2-term in static normal form, and e and e' are not distinguished by any dynamic interpretation $\mathcal{I}^\mathfrak{d}$ of $\Sigma^\mathfrak{d}$, i.e., $\forall\mathcal{I}^\mathfrak{d}.[\![\mathsf{e}]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}} = [\![\mathsf{e}']\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}}$; in other words, term e' and term e have the same (parameterized) meaning. Notice that *NF* is usually partial, since terms for which the static computation diverges have no normal form.

Normalization by Evaluation In this framework, NbE can be described as a technique to write the static normalization function *NF* for a two-level language PL^2 by reduction to evaluation in the ordinary language *PL*. For this to be possible, we require language *PL* to be equipped with a base type `Exp` for code representation of its own terms (and thus of static normal forms in PL^2), and constants that support code construction and name generation.

Filinski shows that in the described setting, NbE can be performed with two type-indexed functions $\downarrow^\tau : \ulcorner\tau\urcorner \to \mathtt{Exp}$ (reification) and $\uparrow_\tau : \mathtt{Exp} \to \ulcorner\tau\urcorner$ (reflection). The function $\downarrow^\tau$ extracts the static normal form of a term $\vdash_{\Sigma^2} \mathsf{e}{:}\tau$ from a special *residualizing instantiation* of the term in *PL*, $\vdash_{\Sigma^{PL}} \ulcorner\mathsf{e}\urcorner{:}\ulcorner\tau\urcorner$; the function $\uparrow_\tau$ is used in both the definition of reification function and the construction of the residualizing instantiation $\ulcorner\mathsf{e}\urcorner$.

Definition 8 (Residualizing Instantiation). *The* residualizing instantiation *of a* Σ^2*-term* $\vdash_{\Sigma^2} \mathsf{e}{:}\tau$ *in PL is* $\vdash_{\Sigma^{PL}} \ulcorner\mathsf{e}\urcorner{:}\ulcorner\tau\urcorner$*, given by* $\ulcorner\mathsf{e}\urcorner = \mathsf{e}\{\Phi_\sqcap\}$ *and* $\ulcorner\tau\urcorner = \tau\{\Phi_\sqcap\}$*, where instantiation* $\Phi_\sqcap$ *is a substitution of* Σ^2*-constructs into* Σ^{PL}*-phrases:* $\Phi_\sqcap(\mathsf{b}^\mathfrak{s}) = \mathsf{b}$*,* $\Phi_\sqcap(\mathsf{b}^\mathfrak{d}) = \mathtt{Exp}$*,* $\Phi_\sqcap(c^\mathfrak{s}) = c$*,* $\Phi_\sqcap(c^\mathfrak{d} : \tau) = \uparrow_\tau \underline{\text{“}c\text{”}}$*,* $\Phi_\sqcap(\$_b) = \mathit{lift}_b$*.*

In words, the residualizing instantiation $\ulcorner\tau\urcorner$ of a fully dynamic type τ substitutes all occurrences of dynamic base types in τ with type `Exp`; since type τ is fully dynamic, type $\ulcorner\tau\urcorner$ is constructed from type `Exp`, and it represents

code values or code manipulation functions (see Section 2.1). The residualizing instantiation $\overline{e}$ of a term e substitutes all the occurrences of dynamic constants and lifting functions with the corresponding code-generating functions. For example, in Examples 4 and 5, if we regard `mult` and `height` as two-level terms, then $\texttt{mult}_\mathsf{r}$ and $\texttt{height}_\mathsf{r}$ are simply their respective residualizing instantiation: $\texttt{mult}_\mathsf{r} = \overline{\texttt{mult}}$ and $\texttt{height}_\mathsf{r} = \overline{\texttt{height}}$.

The function *NF* in NbE is defined by Equation (1) in Figure 5: it computes the static normal form of term e by evaluating the Σ^{PL}-term $\vdash_{\Sigma^{PL}} \downarrow^\tau \overline{e} : \texttt{Exp}$ using an evaluator for language *PL*. In Filinski's semantic framework for TDPE, a correctness theorem of NbE has the following form, though the exact definition of function *NF* varies depending on the setting.

Theorem 9 (Filinski). *The function NF defined in Equation (1) is a static normalization function. That is, for all well-typed Σ^2-terms e, if $e' = NF(e)$, then term e' is in static normal form, and $\forall \mathcal{I}^\mathfrak{d}.[\![e]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}} = [\![e']\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}}$.*

The detail of how Theorem 9 is proved is out of the scope of this article. 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 9.

Normalization by Evaluation

For term $\vdash_{\Sigma^2} e : \tau$, we use

$$NF(e) = [\![\downarrow^\tau \overline{e}]\!]^{\mathcal{I}^{PL}}. \tag{1}$$

to compute its static normal form, where

1. Term $\vdash_{\Sigma^{PL}} \overline{e} : \overline{\tau}$ is the residualizing instantiation of term e, and
2. Term $\vdash_{\Sigma^{PL}} \downarrow^\tau : \overline{\tau} \to \texttt{Exp}$ is the reification function for type τ.

Binding-time annotation The task is, given $\vdash_{\Sigma^{PL}} t{:}\sigma$ and some binding-time constraints, to find $\vdash_{\Sigma^2} t_\mathsf{ann} : \tau$ that satisfies the constraints and

$$[\![\tau]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}_\mathsf{ev}} = [\![\sigma]\!]^{\mathcal{I}^{PL}} \quad , \quad [\![t_\mathsf{ann}]\!]^{\mathcal{I}^\mathfrak{s},\mathcal{I}^\mathfrak{d}_\mathsf{ev}} = [\![t]\!]^{\mathcal{I}^{PL}}$$

Figure 5: A formal recipe for NbE

Partial Evaluation Given a Σ^2-term $\vdash_{\Sigma^2} p : \tau_S \times \tau_D \to \tau_R$, and its static input $\vdash_{\Sigma^2} s : \tau_S$, where both type τ_D and type τ_R are fully dynamic, specialization can be achieved by applying NbE (Equation (1)) to statically normalize the trivial specialization $\lambda x.p(s, x)$:

$$\begin{aligned} NF(\lambda x.p(s,x)) &= [\![\downarrow^{\tau_D \to \tau_R} \overline{\lambda x.p(s,x)}]\!]^{\mathcal{I}^{PL}} \\ &= [\![\downarrow^{\tau_D \to \tau_R} \lambda x.\overline{p}(\overline{s}, x)]\!]^{\mathcal{I}^{PL}} \end{aligned} \tag{2}$$

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 Σ^{PL}-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 interpretation $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}}$ of term t_{ann} agrees with the meaning $[\![t]\!]^{\mathcal{I}^{PL}}$ of term t, and (2) term t_{ann} is compatible with the input's binding-time information in the following sense: combining t_{ann} with the static input results in a term of fully dynamic 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 Σ^{PL}-term $\vdash_{\Sigma^{PL}} p : \sigma_S \times \sigma_D \to \sigma_R$ 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 Σ^2-type τ_S such that $\widetilde{\tau_S} = \sigma_S$; for the more familiar first-order case, type σ_S is some base type b, and type τ_S is simply b^{s}. We need to find a two-level term $\vdash_{\Sigma^2} p_{\mathsf{ann}} : \tau_S \times \tau_D \to \tau_R$, such that (1) types τ_D and τ_R are the fully dynamic versions of types σ_D and σ_R, and (2) $[\![p_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![p]\!]^{\mathcal{I}^{PL}}$.

When given an annotated static input which has the specified binding-time information, $s_{\mathsf{ann}} : \tau_S$ (of some $s : \sigma_S$ such that $[\![s_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![s]\!]^{\mathcal{I}^{PL}}$), we can form the two-level term $\vdash_{\Sigma^2} t_{\mathsf{ann}} \triangleq \lambda x.p_{\mathsf{ann}}(s_{\mathsf{ann}}, x) : \tau_D \to \tau_R$. It corresponds to a one-level term $t \triangleq \lambda x.p(s, x)$, for which (by compositionality of the meaning functions) $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$. Our goal is to normalize term t. If term $e = NF(t_{\mathsf{ann}})$ is the result of the NbE algorithm, we see that its one-level representation $\widetilde{e}$, which we regard as the result of the specialization, has the same meaning as the term t:

$$[\![\widetilde{e}]\!]^{\mathcal{I}^{PL}} = [\![e]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$$

This verifies the correctness of the specialization.

This process of binding-time annotation can be achieved mechanically or manually. In general, one tries to reduce occurrences of dynamic constants in term t, so that more static computation involving constants is carried out during static normalization.

Our setting In this article, the language PL we will work with is essentially ML, with a base type `Exp` for encoding term representations, its associated constructors, constants for name generations (`GENSYM.init` and `GENSYM.new`), and control operators. All of these can be introduced into ML as user-defined data types and functions; in practice, we do not distinguish PL and ML. The associated two-level language PL^2 is constructed from language PL mechanically.

Instantiation through functors Given a two-level term p, the ML module system makes it possible to encode p such that one can switch

between the evaluating instantiation $\widetilde{p}$ and the residualizing instantiation $\overline{p}$ in a structured way [8]. This is done by defining p inside a functor `p_pe(structure D: DYNAMIC) = ...` which parameterizes over all dynamic types, dynamic constants and lifting functions. Depending on how `D` is instantiated, one can create either the evaluating instantiation of p or its residualizing instantiation.

```
signature DYNAMIC =                                          (* Σ^∂ *)
  sig
    type Real                                              (* real^∂ *)

    val mult: Real -> Real -> Real                         (* mult^∂ *)
    val lift_real: real -> Real                            (* $_real *)
  end

functor height_pe(structure D: DYNAMIC) =
 struct
   fun height a z = D.mult (D.lift_real (sin a)) z
 end


structure EDynamic: DYNAMIC =        (* Evaluating Inst. Φ_~ on Σ^∂ *)
  struct
    type Real = real
    fun mult x y  =  x * y
    fun lift_real r = r
  end

structure RDynamic: DYNAMIC =      (* Residualizing Inst. Φ_¬ on Σ^∂ *)
  struct
    local
      open EExp pNbe
      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 *)
structure Rheight = height_pe (structure D = RDynamic);
                                                      (* ¬height *)
```

Figure 6: Instantiation via functors

Example 10. *In Example 5 we sketched how a function* `height` *can be partially evaluated with respect to its first argument. Figure 6 shows how to provide a residualizing instantiation in ML using functors. A functor* `height_pe(structure D:DYNAMIC)` *parameterizes over all dynamic types, dynamic constants and lifting functions that appear in (a two-level version) of* `height`*. The functor* `height_pe` *can be instantiated to yield either the evaluating instantiation* $\widetilde{\texttt{height}}$ *or the residualizing instantiation* $\overline{\texttt{height}}$*.*

3 Formulating Self-application

In this section, we formulate two forms of self-application for TDPE: one simply generates more efficient reification and reflection functions for a type τ; the other adapts the second Futamura projection to generate efficient generating extensions.

Visualization The first natural candidates for considerations of self-applying TDPE are the reification and reflection functions. For example, for a specific fully dynamic type τ, the reification function $\downarrow^\tau$ contains one β-redex for each recursive call following the type structure. We want to use TDPE to remove the overhead of β-reductions.

Starting from $\vdash_{\Sigma^{PL}} \downarrow^\tau : \overline{\tau} \to \texttt{Exp}$, we follow the "recipe" outlined in Figure 5 to achieve specialization. First, the term needs to be binding-time annotated. A straightforward analysis of the implementation of NbE (see Figures 3 and 4) shows that all the base types (`Exp`, `Var`, etc.) and constants (`APP`, `Gensym.init`, etc.) are needed in the code generation phase; hence they all should be classified as dynamic. We thus introduce a trivial binding-time annotation.

Definition 11 (Trivial Binding-Time Annotation). *The* trivial binding-time annotation *of a Σ^{PL}-term $\vdash_{\Sigma^{PL}} t : \sigma$ is a PL^2-term $\vdash_{\Sigma^2} \underline{\langle t \rangle} : \underline{\langle \sigma \rangle}$, given by $\underline{\langle t \rangle} = t\{\Phi_{\underline{\ }}\}$ and $\underline{\langle \sigma \rangle} = \sigma\{\Phi_{\underline{\ }}\}$, where the instantiation $\Phi_{\underline{\ }}$ is a substitution of Σ^{PL}-constructs into Σ^2-phrases: $\Phi_{\underline{\ }}(\mathsf{b}) = \mathsf{b}^{\mathfrak{d}}$, $\Phi_{\underline{\ }}(\ell : \mathsf{b}) = \$_{\mathsf{b}}\ell^{\mathfrak{s}}$ (ℓ is a literal), $\Phi_{\underline{\ }}(c) = c^{\mathfrak{d}}$ (c is not a literal).*

The following properties hold:

1. $[\![\underline{\langle t \rangle}]\!]^{\mathcal{I}^{\mathfrak{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$, *making* $\underline{\langle t \rangle}$ *a binding-time annotation of* t*;*
2. $\widetilde{\underline{\langle t \rangle}} = t$*;*
3. $\underline{\langle \sigma \rangle}$ *is always a fully dynamic type;*
4. *If a Σ^2-type τ is fully dynamic, then* $\overline{\underline{\langle \overline{\tau} \rangle}} = \overline{\tau}$*.*

Since the trivial binding-time annotation of $\downarrow^\tau$, $\vdash_{\Sigma^2} \underline{\langle \downarrow^\tau \rangle} : \underline{\langle \overline{\tau} \to \texttt{Exp} \rangle}$, has a fully dynamic type, we can apply NbE (Equation (1)) to achieve static

normalization for this term:

$$NF(\underline{\langle \downarrow^\tau \rangle}) = [\![\downarrow^{\underline{\overline{\tau} \to \mathtt{Exp}\rangle}} (\overline{\underline{\langle \downarrow^\tau \rangle}})]\!]^{\mathcal{I}^{PL}}.$$

We write $\Downarrow^\tau$ for $\overline{\underline{\langle \downarrow^\tau \rangle}}$ and $\Uparrow_\tau$ for $\overline{\underline{\langle \uparrow_\tau \rangle}}$ for notational conciseness. Meanwhile, the reader should keep in mind that, by definition, they are just the residualizing instantiation of the two-level terms $\underline{\langle \downarrow^\tau \rangle}$ and $\underline{\langle \uparrow_\tau \rangle}$, respectively. In fact, term $\downarrow^\tau$ and term $\Downarrow^\tau$ are respectively the evaluating instantiation and residualizing instantiation of the same two-level term $\underline{\langle \downarrow^\tau \rangle}$: $\widetilde{\underline{\langle \downarrow^\tau \rangle}} = \downarrow^\tau$, and $\overline{\underline{\langle \downarrow^\tau \rangle}} = \Downarrow^\tau$; analogous for term $\uparrow_\tau$ and term $\Uparrow_\tau$. We will exploit this fact in Section 4.1 to apply the functor-based approach to the reification/reflection combinators themselves.

Since $\underline{\langle \cdot \rangle}$ and $\overline{\cdot}$ are simply substitutions, they distribute over all type constructors. Note also that both $\downarrow^\tau$ and $\uparrow_\tau$ depend only on $\overline{\tau}$; that is, if $\overline{\tau_1} = \overline{\tau_2}$, then $\downarrow^{\tau_1} = \downarrow^{\tau_2}$ and $\uparrow_{\tau_1} = \uparrow_{\tau_2}$. This justifies our practice of writing "$\bullet$" to denote arbitrary dynamic base type in τ. A simple derivation using these properties and property (3) in Definition 11 gives the following simple formulation of the self-application.

$$NF(\underline{\langle \downarrow^\tau \rangle}) = [\![\downarrow^{\tau \to \bullet} (\Downarrow^\tau)]\!]^{\mathcal{I}^{PL}}. \tag{3}$$

The following corollary follows immediately from Theorem 9 and the property of trivial binding-time annotation.

Corollary 12. *If $\mathsf{e}_\tau = NF(\underline{\langle \downarrow^\tau \rangle})$, then its one-level representation $\widetilde{\mathsf{e}_\tau}$ is free of β-redexes and is semantically equivalent to $\downarrow^\tau$:*

$$[\![\widetilde{\mathsf{e}_\tau}]\!]^{\mathcal{I}^{PL}} = [\![\mathsf{e}_\tau]\!]^{\mathcal{I}^{\mathsf{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\underline{\langle \downarrow^\tau \rangle}]\!]^{\mathcal{I}^{\mathsf{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![\downarrow^\tau]\!]^{\mathcal{I}^{PL}}$$

Normalizing $\downarrow^\tau$ not only speeds up the reification, but also helps one to, as Danvy phrased it [6], *visualize* reification for a particular type. Indeed, the reification and reflection functions of Examples 2 and 3 are presented in the normalized form, which can be generated by this visualization.

Danvy carried out a similar self-application in the setting of Scheme [6]; 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 $\overline{\underline{\langle c \rangle}} = \overline{c^{\mathfrak{d}}} = \uparrow_{\underline{\langle \sigma \rangle}} \underline{\text{"}c\text{"}}$. Consequently the results of these two treatments are essentially the same.

Adapted second Futamura projection As we have argued in the introduction, in the setting of TDPE, following the second Futamura projection literally is not a viable choice for deriving efficient generating extensions—the evaluator for language *PL*, which implements the meaning function $[\![\cdot]\!]^{\mathcal{I}^{PL}}$, might not even be written in *PL*; making such an evaluator explicit in the partial evaluator to be specialized will bring in a large overhead, 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 'trivial' generating extension (usually) yields a more efficient generating extension.

Recall from Equation (2) that in order to specialize a two-level term $\vdash_{\Sigma^2} \mathsf{p} : \tau_S \times \tau_D \to \tau_R$ with respect to a static input $\vdash_{\Sigma^2} \mathsf{s} : \tau_S$, we execute the Σ^{PL}-program $\vdash_{\Sigma^{PL}} \downarrow^{\tau_D \to \tau_R} \lambda d.\overline{\mathsf{p}}(\overline{\mathsf{s}}, d) : \mathtt{Exp}$. By λ-abstracting over the residualizing instantiation $\overline{\mathsf{s}}$ of the static input s, we can trivially obtain a generating extension $\mathsf{p}^\dagger$, which we will refer to as the trivial generating extension.

$$\vdash_{\Sigma^{PL}} \mathsf{p}^\dagger \triangleq \lambda s_\mathsf{r}.\downarrow^{\tau_D \to \tau_R} (\lambda d.(\overline{\mathsf{p}}(s_\mathsf{r}, d))) : \overline{\tau_S} \to \mathtt{Exp}.$$

Corollary 13. *The term $\mathsf{p}^\dagger$ is a generating extension of program p.*

Since the term $\mathsf{p}^\dagger$ is itself a Σ^{PL}-term, we can follow the recipe in Figure 5 to specialize it into a more efficient generating extension. There might be some flexibility in the binding-time annotation of the term $\overline{\mathsf{p}}$. We therefore analyze the different occurrences of constants in $\overline{\mathsf{p}}$. Since $\overline{\cdot} = \Phi_{\lceil\rceil}$ is an instantiation, i.e., a substitution on dynamic constants and lifting functions, every constant c' in $\overline{\mathsf{p}}$ must appear as a subterm of the image of a constant or a lifting function under the substitution $\Phi_{\lceil\rceil}$. If c' appears inside $\Phi_{\lceil\rceil}(c^\mathsf{D}) = \uparrow_\tau \underline{\text{“}c\text{”}}$ (where c' could be a code-constructor such as `LAM`, `APP` appearing in term $\uparrow_\tau$) , or $\Phi_{\lceil\rceil}(\$_\mathsf{b}) = \mathit{lift}_\mathsf{b}$, then c' is needed in the code generation phase, and hence it should be classified as dynamic. If c' appears inside $\Phi_{\lceil\rceil}(c^\mathsf{S}) = c$, 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 $\mathsf{p}^\dagger$, since the input s is not statically available at this stage. It is thus not too conservative to take the trivial binding time annotation of the trivial generating extension $\mathsf{p}^\dagger$, and proceed with Equation (1) to generate a more efficient generating extension.

$$\begin{aligned}
\mathsf{p}^\ddagger &= NF(\langle \underline{\lambda s_\mathsf{r}.\downarrow^{\tau_D \to \tau_R} (\lambda d.(\overline{\mathsf{p}}\ (s_\mathsf{r}, d)))} \rangle) \\
&= [\![\downarrow^{\langle \tau_S \to \bullet \rangle} \overline{\langle \underline{\lambda s_\mathsf{r}.\downarrow^{\tau_D \to \tau_R} (\lambda d.(\overline{\mathsf{p}}\ (s_\mathsf{r}, d)))} \rangle}]\!]^{\mathcal{I}^{PL}} \\
&= [\![\downarrow^{\tau_S \to \bullet} (\lambda s_\mathsf{r}.\overline{\langle \underline{\downarrow^{\tau_D \to \tau_R}} \rangle} (\lambda d.(\overline{\langle \underline{\overline{\mathsf{p}}} \rangle}\ (s_\mathsf{r}, d)))))]\!]^{\mathcal{I}^{PL}}
\end{aligned}$$

Writing $\overline{\mathsf{p}}$ for the subterm $\overline{\langle \underline{\overline{\mathsf{p}}} \rangle}$, we have

$$\mathsf{p}^\ddagger = [\![\downarrow^{\tau_S \to \bullet} (\lambda s_\mathsf{r}.\Downarrow^{\tau_D \to \tau_R} (\lambda d.\overline{\mathsf{p}})\ (s_\mathsf{r}, d))]\!]^{\mathcal{I}^{PL}} \tag{4}$$

The correctness of the second Futamura projection follows from Corollary 13 and Theorem 9.

Corollary 14. *Program $\widetilde{\mathsf{p}^\ddagger}$ (the one-level form of the static normal form $\mathsf{p}^\ddagger$) is a generating extension of p which is free of β-redexes.*

Proof. By Theorem 9 and the property of trivial binding-time analysis, we have $p^\ddagger$ is in static normal form, and $[\![\widetilde{p^\ddagger}]\!]^{\mathcal{I}^{PL}} = [\![p^\dagger]\!]^{\mathcal{I}^{PL}}$. That the program $\widetilde{p^\ddagger}$ is a generating extension of p follows from Corollary 13. □

Now let us examine how the term $\ulcorner\ulcorner p\urcorner\urcorner$ is formed. Note that $\ulcorner\ulcorner p\urcorner\urcorner = \overline{\langle \ulcorner p \urcorner \rangle} = ((p\{\Phi_{\sqcap}\})\{\Phi_{\langle\rangle}\})\{\Phi_{\sqcap}\} = p\{\Phi_{\sqcap} \circ \Phi_{\langle\rangle} \circ \Phi_{\sqcap}\}$; thus $\ulcorner\cdot\urcorner$ corresponds to the composition of three instantiations, $\Phi_{\mathtt{m}} = \Phi_{\sqcap} \circ \Phi_{\langle\rangle} \circ \Phi_{\sqcap}$, which is also an instantiation. We call $\Phi_{\mathtt{m}}$ the generating-extension instantiation (GE-instantiation); a simple calculation gives its definition.

Definition 15 (GE-instantiation). *The* GE-instantiation *of a* Σ^2*-term* $\vdash_{\Sigma^2} e : \tau$ *in PL is* $\vdash_{\Sigma^{PL}} \ulcorner\ulcorner e\urcorner : \ulcorner\tau\urcorner\urcorner$ *given by* $\ulcorner\ulcorner e\urcorner\urcorner = e\{\Phi_{\mathtt{m}}\}$ *and* $\ulcorner\tau\urcorner = \tau\{\Phi_{\mathtt{m}}\}$*, where instantiation* $\Phi_{\mathtt{m}}$ *is a substitution of* Σ^2*-constructs into* Σ^{PL}*-phrases:*

$$
\begin{aligned}
\Phi_{\mathtt{m}}(\mathsf{b}^{\mathsf{s}}) &= \Phi_{\mathtt{m}}(\mathsf{b}^{\mathsf{d}}) = \mathtt{Exp} \\
\Phi_{\mathtt{m}}(c^{\mathsf{s}} : \tau) &= \overline{\langle \underline{c : \ulcorner\tau\urcorner} \rangle} = \uparrow_{\langle \ulcorner\tau\urcorner \rangle} \underline{\text{“}c\text{”}} \\
\Phi_{\mathtt{m}}(c^{\mathsf{d}} : \tau) &= \overline{\uparrow_\tau \underline{\text{“}c\text{”}}} = \Uparrow_\tau \overline{\langle \underline{\mathtt{Var}} \rangle} (\mathit{lift}_{\mathtt{string}} \text{“}c\text{”}) \\
\Phi_{\mathtt{m}}(\$_b) &= \uparrow_{\bullet\to\bullet} \underline{\text{“}\mathit{lift}_b\text{”}}
\end{aligned}
$$

Note that at some places, we intentionally keep the $\ulcorner\cdot\urcorner$ 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 4.5, we show how to extend the instantiation-through-functor approach to cover GE-instantiation as well.

4 The Implementation

In this section we treat various issues arising when implementing in ML the abstract formulation given in Section 3.

4.1 Residualizing Instantiation of the Combinators

In Section 3 we remarked that $\downarrow^\tau$ and $\Downarrow^\tau$ are respectively the evaluating instantiation and the residualizing instantiation of the same two-level term $\langle \underline{\downarrow^\tau} \rangle$. This suggests that the ML module system can be used again to conveniently implement both instantiations for a given two-level term (see Example 10). However, we have formulated reification and reflection as type-indexed functions, and implement them not as a monolithic program, but as combinators which can be plugged together by the user to construct a type encoding as a reification-reflection pair $(\downarrow^\tau, \uparrow_\tau)$. Fortunately, because $\langle \underline{\cdot} \rangle$ is a substitution, it distributes over all constructs in a term; it marks all the types and constants as dynamic. Therefore it suffices to parameterize

all the combinators. These combinators, when instantiated with either an evaluating or a residualizing interpretation, can be combined according to a type τ to yield either $(\downarrow^\tau, \uparrow_\tau)$ or $(\Downarrow^\tau, \Uparrow_\tau)$.

We basically already have achieved our goal, because the functor `nbe` (see Figure 3) is parameterized over the primitives used in the NbE module. Hence, rather than hardwiring code-generating primitives, this factorization reuses the implementation for producing both the evaluating instantiation and the residualizing instantiation. An evaluating instantiation `ENbe` of NbE is produced by applying the functor `nbe` to the standard evaluating structures `EExp`, `EGensym` and `ECtrl` of the signatures `EXP`, `GENSYM` and `CTRL`, respectively (Figure 7—we show the implementations of structures `EExp` and `EGensym`; for structure `ECtrl`, we use Filinski's implementation [12]). Residualizing instantiations `RNbe` of Full NbE and `RpNbE` of Pure NbE result from applying the functors `nbe` and `pureNbe`, respectively, to appropriate residualizing structures `RGensym`, `RExp`, and `RCtrl` (Figure 8).

```
structure EExp                                  (* Evaluating Inst. Φ~ 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. Φ~ 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 ENbe = nbe (structure G = EGensym
                      structure E = EExp
                      structure C = ECtrl): NBE
```

Figure 7: Evaluating Instantiation of NbE

```
structure RExp: EXP = struct
  type Exp = EExp.Exp
  type Var = EExp.Exp

  (* VAR v = APP@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 RNbe = nbe (structure G = RGensym
                      structure E = RExp
                      structure C = RCtrl): NBE

structure RpNbe = pureNbe (structure G = RGensym
                           structure E = RExp): NBE
```

Figure 8: Residualizing Instantiation of NbE

For example, in the structure `RExp`, the type `Exp` and the type `Var` are both 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 (visualized) the definition of reflection (see also Example 3).

With the residualizing interpretation of reification and reflection at our disposal, we now can perform visualization by following Equation (3).

Example 16. *We show the visualization of* $\downarrow^{\bullet\to\bullet}$ *for Pure NbE. Following Equation (3), we have to compute* $\downarrow^{(\bullet\to\bullet)\to\bullet}$ $(\Downarrow^{\bullet\to\bullet})$. *This is done in Figure 9; it is not difficult to see that the result matches the execution of the term* `reify (a' --> a')` *(see Figure 4). Visualization of the reflection function is carried out similarly.*

4.2 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

```
local open ENbe
  infixr 5 -->
  val Ereify_aa_exp
    = reify ((a' --> a') --> a')                    (* ↓^((•→•)→•) *)
  open RpNbe
  infixr 5 -->
  val Rreify_aa = reify (a' --> a')                 (* ⇓^(•→•) *)
in val _ = Ereify_aa_a (Rreify_aa) end
```

The (pretty-printed) result is:

$$\begin{array}{l}\lambda x_1.\mathbf{let}\ r_2 = \mathsf{init}() \\ \qquad\qquad r_3 = \mathsf{new}() \\ \quad \mathbf{in} \\ \qquad \underline{\lambda\, r_3}.x_1\ \underline{r_3}\end{array}$$

Figure 9: Visualizing $\downarrow^{\bullet\to\bullet}$

numeral 0_{ch}, successor s_{ch}, and the addition function $+_{\mathsf{ch}}$ in Figure 10 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 $\overline{\text{"}+_{\mathsf{ch}}\text{"}} = \overline{+_{\mathsf{ch}}} = +_{\mathsf{ch}}$. Following Equation (4), we can compute an efficient generating extension $+_{\mathsf{ch}}{}^{\ddagger}$, as shown in Figure 10.

4.3 Type Specification For Self-Application

When performing visualization in Figure 9, we followed Equation (3) step by step, building $\downarrow^{(\bullet\to\bullet)\to\bullet}$ and $\Downarrow^{\bullet\to\bullet}$ with two sets of combinators, one for the residualizing instantiation of NbE, the other for the evaluating instantiation. Notice that `a' --> a'` is constructed twice, once with each set of combinators—when visualizing $\downarrow^\tau$ by coding Equation (3) by hand, τ always has to be encoded twice. This is cumbersome; more important, good engineering practice demands the abstract formulation of various instances of self-application to be programs by themselves, instead of templates to be instantiated manually. That type-indexed functions are implemented as combinators makes it impossible to abstract over the type argument at a function level. Because the problem appears as a result of the interaction between types, we should first analyze the necessary type structure.

In both forms of the self applications, types are used to index several different families of type-indexed values $\{\downarrow^\tau\}_\tau$, $\{\Downarrow^\tau\}_\tau$ and $\{\Uparrow_\tau\}_\tau$; in particular,

```
type 'a num = ('a -> 'a) -> ('a -> 'a)                    (* Type num *)
val c0 : EExp.Exp num
  = fn f => fn x =>  x                                    (* ⌜0ch⌝ : ⌜num⌝ *)
fun cS (n: EExp.Exp num)
  = fn f => fn x => f (n f x)                             (* ⌜sch⌝ : ⌜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 ENbe
  infixr 5 -->
  val Ereify_n_exp
  = reify  (((a' --> a') --> (a' --> a')) --> a')
                                                          (* ↓⌜num⌝→• *)
  open RpNbe
  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:

$$\begin{array}{l}\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}()\ \ r_3 = \mathsf{new}()\ \ r_4 = \mathsf{new}()\\ \qquad\qquad r_5 = \mathsf{new}()\ \ r_7 = \mathsf{new}()\\ \quad\ \textbf{in}\\ \qquad \underline{\lambda}\,\underline{r_3}.\underline{\lambda}\,\underline{r_4}.\underline{\lambda}\,\underline{r_5}.(x_1(\lambda\, x_6.(\underline{r_4}\,\underline{@}\,x_6)))\\ \qquad\qquad\qquad\qquad (((\underline{r_3}\,\underline{@}\,(\underline{\lambda}\,\underline{r_7}.(\underline{r_4}\,\underline{@}\,\underline{r_7})))\,\underline{@}\,\underline{r_5})))\end{array}$$

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 10: Church numerals

the same type might be used to index both families of type-indexed values. Our method of implementing type-indexed values is to plug together combinators that correspond to the type-indexed values according to the type structure. Therefore, specifying a type amounts to writing down how combinators are plugged together, parameterized over the definition of the combinators, i.e., an `Nbe`-structure. A self-application algorithm should take such parameterized type encodings, instantiate them with the combinators

```
signature VIS_INPUT =                (* Signature for a type specification *)
  sig
    type 'a vis_type          (* Type τ, 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 = P.inp(sNbe)        (* Instantiating with sNbe *)
      structure rVIS = P.inp(dNbe)        (* Instantiating with dNbe *)
      open sNbe
      infixr 5 -->
    in
      val vis = ENbe (eVIS.T_enc --> a') (RpNbe.nbe rVIS.T_enc)
                                                    (* ↓^τ → • (⇓^τ) *)
    end
  end
```

Figure 11: Specifying types as functors

for $\{\downarrow^\tau\}_\tau$, $\{\Downarrow^\tau\}_\tau$ and $\{\Uparrow_\tau\}_\tau$, respectively, and combine the result according to the corresponding formulation (Equation (3) or Equation (4)).

Consider the example of visualizing reification: the specification of a type τ should consists not only the type τ itself, but also a functor that maps a NBE-structure Nbe to the pair $(\downarrow^\tau, \uparrow_\tau)$, which is of type τ Nbe.rr. This suggests that the type specification should have the following dependent type:

$$\sum \tau : *. \prod \texttt{Nbe} : \texttt{NBE}.(\tau\ \texttt{Nbe}.rr),$$

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 11).

The example visualization in Figure 9 can be now carried out using the type specification given in Figure 12.

4.4 Monomorphizing Control Operators

Let-Insertion via Control Operators

```
structure a2a : VIS_INPUT =                    (* A type specification *)
  struct
    type 'a vis_type = 'a -> 'a                          (* τ = • → • *)
    functor inp(Nbe: NBE) =                                    (* Nbe *)
      struct
        val T_enc = Nbe.--> (Nbe.a', Nbe.a')            (* τ Nbe.rr *)
      end
    end

structure vis_a2a = vis_reify(a2a);                  (* Visualization *)
```

Figure 12: Type specification for visualizing $\downarrow^{\bullet\to\bullet}$

Full TDPE deals with cbv-languages with computational effects. In such a setting, a standard partial-evaluation technique to prevent duplicating or discarding computations that have side-effects is *let-insertion* [2, 16]: all computation that might have effects will be bound to a variable and sequenced using the (monadic) let construct. However, when the TDPE algorithm identifies the need to insert a let-construct, the execution usually is not at a point where a let-construct can be inserted, i.e., a code-generating expression.

Danvy [5] solves this problem by using Danvy and Filinski's control operators shift and reset: roughly speaking, operator shift grabs the current evaluation context up to the closest delimiter reset and passes it to its argument, which can then invoke this delimited evaluation context just like a normal function. Formally, Danvy and Filinski introduced the semantics of shift and reset in terms of the CPS transformation (Figure 13; see Danvy and Filinski [9] and Filinski [12] for more details).

$$
\begin{array}{rcl}
[\![\mathsf{shift}\ E]\!]_{\mathsf{CPS}} & = & \lambda\kappa.[\![E]\!]_{\mathsf{CPS}}(\lambda f.f(\lambda v.\lambda\kappa'.\kappa'(\kappa\ v))(\lambda x.x)) \\
[\![\mathsf{reset}\langle E\rangle]\!]_{\mathsf{CPS}} & = & \lambda\kappa.\kappa([\![E]\!]_{\mathsf{CPS}}(\lambda x.x))
\end{array}
$$

Figure 13: The CPS semantics of shift/reset

With the help of these control operators, Danvy's treatment [5] achieves the task of let-insertion as follows: (1) use a reset to surround every expression that has type `Exp`, thus 'marking the boundaries' for code generation; (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 reification and reflection functions given by Danvy are shown in Figure 14; there are two function type constructors: function type without effects $\tau_1 \to \tau_2$, which does not need let-insertion, and function type with possible latent effects $\tau_1 \xrightarrow{!} \tau_2$, which performs let-insertion (we extend the type `Exp` of code representations with a constructor `LET of string * Exp * Exp` and write $\underline{\mathbf{let}}\ \underline{x} = e_1\ \underline{\mathbf{in}}\ e_2$ for `LET (`x`,`e_1`,`e_2`)`; we implement a new TDPE constructor `-!>` for the new type constructor $\xrightarrow{!}$).

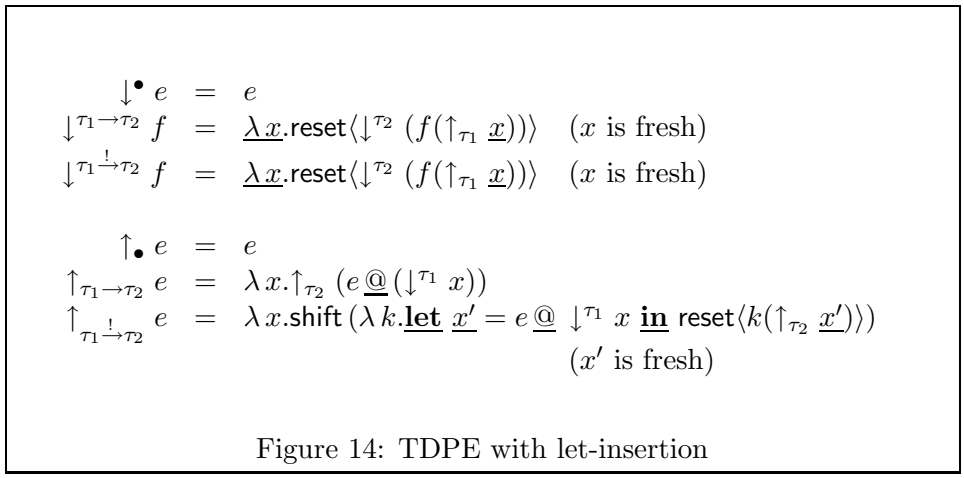

Figure 14: TDPE with let-insertion

Monomorphizing control operators In the definition of reflection for function types with latent effects, $\uparrow_{\tau_1 \xrightarrow{!} \tau_2}$, the return type (here τ_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 $\uparrow_{\tau_2} x'$; this prompts us to move this term into the context surrounding the shift-expression, and to apply k to a simple unit value ()—no information needs to be carried around, except for the transfer of the control flow.

$$\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2} e = \lambda x.(\lambda\,().\ \uparrow_{\tau_2} \underline{x'})(\mathsf{shift}\,(\lambda k.\underline{\mathbf{let}}\ \underline{x'} = e\,\underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ \mathsf{reset}\langle k()\rangle)) \quad (x' \text{ is fresh})$$

Now the aforementioned problem is solved, since the return type of shift is fixed to unit.

Recently, Sumii (email exchange, February 2000) pointed out that the reset in the above definition can be removed. This simplification improves the performance of TDPE and the generated extension to be generated by

self-application.

$$\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2} \; e = \lambda x.(\lambda\,().\ \uparrow_{\tau_2} \underline{x'})(\mathsf{shift}\,(\lambda k.\underline{\mathbf{let}}\ \underline{x'} = e\,\underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ k())) \qquad (x' \text{ is fresh})$$

To show this change is semantic-preserving, we compare the CPS-based denotational semantics of both the original definition and the new definitions.

Proposition 17. *The terms* $[\![\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$, $[\![\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta\eta$*-convertible.*

Example 18. *The monomorphized definition of reflection for function types with latent effects is amenable to TDPE itself. Figure 15 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 appears to be much compact.*

Sum types Full TDPE also deals with sum types using control operators; this treatment is also due to Danvy [6]. Briefly, the operator shift is used in the definition of reflection function for sum types, $\uparrow_{\tau_1+\tau_2}$. As the type suggests, the return type of this function should be a value of type $\ulcorner\tau_1\urcorner + \ulcorner\tau_2\urcorner$, i.e., a value of the form **inl** $(v_1 : \ulcorner\tau_1\urcorner)$ or **inr** $(v_2 : \ulcorner\tau_2\urcorner)$ (for some appropriate v_1 or v_2), but not both; on the other hand, both values are needed to have the complete information. The solution by Danvy is to "return twice" to the context by capturing the delimited context and apply it separately to **inl** $(\uparrow_{\tau_1} e_1)$ and **inr** $(\uparrow_{\tau_2} e_2)$, and combine the result using a case-construct which introduces the bindings for e_1 and e_2. The original definition of $\uparrow_{\tau_1+\tau_2}$ is given below.

$$\begin{aligned}\uparrow_{\tau_1+\tau_2} e = \mathsf{shift}(\lambda k.\ \underline{\mathbf{case}}\ e\ \underline{\mathbf{of}}\ &\underline{\mathbf{inl}}(\underline{x_1}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inl}\ (\uparrow_{\tau_1} \underline{x_1}))\rangle \\ \mid\ &\underline{\mathbf{inr}}(\underline{x_2}) \Rightarrow \mathsf{reset}\langle k(\mathbf{inr}\ (\uparrow_{\tau_2} \underline{x_2}))\rangle\ )\end{aligned} \qquad (x_1, x_2 \text{ are fresh})$$

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 for the sake of self-application.

Following the same analysis as before, we observe that the arguments to k must be one of the two possibilities, **inl** $(\uparrow_{\tau_1} e_1)$ and **inr** $(\uparrow_{\tau_2} e_2)$: 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 booleans **true** and

Visualization of $\downarrow^{(\bullet \xrightarrow{!} \bullet) \xrightarrow{!} \bullet}$ results in:

```
λx1.let
      r2 = init()
      r3 = new()
      r11 = reset⟨ let
                     r10 = x1(λx5.let
                                    r6 = new()
                                    r9 = shift(λx7.let
                                                     r8 = x7()
                                                   in
                                                    let r6 = r3 @ x5 in r8
                                                   end)
                                  in
                                   r6
                                  end)
                   in
                    r10
                   end⟩
   in
    λ r3.r11
   end
```

In contrast, visualization of $\downarrow^{(\bullet \to \bullet) \to \bullet}$ of Pure NbE results in:

```
λx1.let r2 = init()
        r3 = new()
    in  λ r3.x1(λx4.r3 @ x4)
    end
```

Figure 15: Visualizing TDPE with let-insertion

false as arguments to continuation k (again, following Sumii's suggestion, we have dropped the unncessary uses of reset):

$$\begin{aligned}\uparrow^{\text{new}}_{\tau_1+\tau_2} e = \quad & \textbf{if } \mathsf{shift}(\lambda k.\ \underline{\textbf{case}}\ e\ \underline{\textbf{of}}\ \underline{\textbf{inl}}(x_1) \Rightarrow k\ \textbf{true} \\ & \qquad\qquad\qquad\qquad \mid\ \underline{\textbf{inr}}(x_2) \Rightarrow k\ \textbf{false}\ ) \\ & \textbf{then inl } (\uparrow_{\tau_1} x_1) \textbf{ else inr } (\uparrow_{\tau_2} x_2) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (x_1, x_2 \text{ are fresh})\end{aligned}$$

The use of shift is instantiated with the fixed boolean type. Again, we check that this change does not modify the semantics.

Proposition 19. $[\![\uparrow^{\text{new}}_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta\eta$*-convertible.*

4.5 Pragmatics

We generalize the technique of encoding a two-level term p in ML presented at the end of Section 2.3: we code p inside a functor

```
p_ge(structure S:STATIC structure D:DYNAMIC) = ...
```

which parameterizes over both static and dynamic constants. With suitable instantiations of `S` and `D` one thus can create not only $\widetilde{p}$ and $\ulcorner p \urcorner$, but also $\ulcorner\!\ulcorner p \urcorner\!\urcorner$; the instantiation table displayed in Figure 16 summarizes how to write the components of the three kinds of instantiation functors for `S` and `D`.

		$\Phi_\sim$	$\Phi_{\ulcorner\urcorner}$	$\Phi_{\ulcorner\!\ulcorner\urcorner\!\urcorner}$
S	b^{s}	b	b	`Exp`
	$c^{\mathsf{s}} : \tau$	c	c	$\uparrow_{\langle \ulcorner \tau \urcorner \rangle}$ "c"
D	b^{d}	b	`Exp`	`Exp`
	$c^{\mathsf{d}} : \tau$	c	$\uparrow_\tau$ "c"	$\Uparrow_\tau \ulcorner \langle \mathtt{Var} \rangle \urcorner (\mathit{lift}_{\mathtt{string}}$ "c")
	$\$_{\mathsf{b}}$	$\lambda x.x$	$\mathit{lift}_{\mathsf{b}}$	$\uparrow_{\bullet\to\bullet}$ "$\mathit{lift}_{\mathsf{b}}$"

Figure 16: Instantiation table

Note, in particular, that $\Phi_\sim$ and $\Phi_{\ulcorner\urcorner}$ have the same instantiation for the static signature; hence we can reuse $\Phi_\sim$ on Σ^{s} for $\Phi_{\ulcorner\urcorner}$ on Σ^{s}.

Example 20. *We revisit the function* `height` *from Examples 5 and 10. In Figure 17 we define the functor* `height_ge` *along with signatures* `STATIC` *and* `DYNAMIC`*. Structure* `GEStatic` *and structure* `GEDynamic` *provide the GE-instantiation for the signature* Σ^2*. The instantiation of* `height_ge` *with these structures gives* $\ulcorner\!\ulcorner \mathtt{height} \urcorner\!\urcorner$*. Applying the second Futamura projection as given in Equation (4) yields*

$$\begin{array}{l}\lambda x_1.\ \textbf{let}\ r_2 = \mathsf{init}() \\ \qquad\qquad r_3 = \mathsf{new}() \\ \quad \textbf{in} \\ \quad\ \underline{\lambda\, r_3.}\ \underline{\text{“}\mathit{mult}\text{”}}\ \underline{@}\ (\mathit{lift}_{\mathsf{real}}(\mathit{sin}\ x_1))\ \underline{@}\ \underline{r_3}\end{array}$$

5 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

```
signature STATIC =                                          (* Σ^s *)
  sig
    type SReal                                             (* real^s *)
    val sin: SReal -> SReal                                 (* sin^s *)
  end

signature DYNAMIC =                                         (* Σ^d *)
  sig
    type SReal                                             (* real^s *)
    type DReal                                             (* real^d *)
    val mult: DReal -> DReal -> DReal                      (* mult^d *)
    val lift_real: SReal -> DReal                          (* $_real *)
  end

functor height_ge(structure S: STATIC
                  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_mm on Σ^s *)
  struct
    local open EExp ENbe; infixr 5 -->
    in
      type SReal = Exp
      val sin = reflect (a' --> a') (VAR "sin")
    end
  end

structure GEDynamic: DYNAMIC =                        (* Φ_mm on Σ^d *)
  struct
    local open RExp RNbe; 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
```

Figure 17: Instantiation via functors

this purpose in several instances [5, 6, 10, 11]. Having implemented the second Futamura projection, we can instead *generate* a compiler, which is the generating extension of an interpreter.

One of the languages for which compiling using TDPE has been stud-

ied [5] is *Tiny* [21], a prototypical imperative language. As outlined in Section 2.2, 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` which is parameterized over all dynamic constructs. Appendix A.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 the trivial generating extension `meaning`†.

Following the development in Section 4.5, we proceed in three steps:

1. Rewrite `tiny_pe` into a 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 Figure 16, thereby creating the GE-instantiation $\overline{\texttt{"meaning"}}$.

3. Perform the second Futamura projection; this yields the efficient generating extension `meaning`‡, a Tiny compiler.

Appendix A.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 steps from above 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 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 instance of a constant appearing in the program, its monotype has to be determined; polymorphic constants have to be monomorphized, possibly creating several instances if the constant is used at more than one type. This is a consequence of performing *type-directed* partial evaluation; for the second Futamura projection, every constant is instantiated with a code-generating 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 separated, 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 which have to be considered, since functions, products and sums do not require binding-time annotations, and

- TDPE utilizes the ML type system: type checking checks the consistency of the binding-time annotations.

6 Benchmarks

In Section 3 we claimed that the 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 in comparison to $p^\dagger$ we ran some benchmarks. These were performed on a 250 MHz Silicon Graphics O_2 workstation using Standard ML of New Jersey version 110.0.3.

To compare the generated compiler `meaning`‡ with the trivial generating extension `meaning`†, we compiled the factorial function written in *Tiny* 1,000,000 times; using `meaning`† it took 261.2 seconds, whereas using `meaning`‡ it took 194.9 seconds—a speedup of about 34%. Notice, however, that the trivial generating extension `meaning`† we used for the benchmarks is based on partially evaluating the fully parameterized version of `meaning` (see Section 5 and Appendix A), i.e., a suitable instantiation of `tiny_ge` rather than `tiny_pe`. Performing the same experiment with the original version `tiny_pe`, where only the dynamic constructs are parameterized over, actually takes only 169.5 seconds—it is even faster than `meaning`‡. The reason for this slowdown is that `meaning`‡ 'inherits' the parameterization from `tiny_ge` as a source of inefficiency. Eliminating the top-level fixed-point operator and the user-defined case operators (see Section A.2.1) in `meaning`‡ by using the built-in recursion and case construct reduced the runtime to 99.2 seconds.[1] If we compare the running times of the generating extensions with 'less parameterization', i.e., `meaning`† based on `tiny_pe` and `meaning`‡ where the user-defined fixed point operator and case operators were removed, the speedup is about 71%.

The speedup of the second Futamura projection with TDPE for compiling *Tiny* programs is disappointing compared to the order-of-magnitude speedup achievable in traditional partial evaluation [19]. 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 any such overhead in the first place, the further speedup is bound to be lower.

[1] Initially, this elimination was carried out by hand. Later, we found out how it can be carried out automatically by (1) incorporating TDPE with patterns as generated bindings; and (2) systematically changing the code-generating instantiations for the fixed point and case operators used. These two improvements are independent of each other. The details of these techniques are out of the scope of the present article.

Which overhead of TDPE can we actually remove by using the second Futamura projection? An examination of the basic algorithm (see Figure 2) shows that the actual work done by TDPE is governed by the type index at which reification is used. Thus one would expect that using the second Futamura projection for specializing programs of higher types gives a better speedup. This is indeed the case: running the generating extension $+_{\mathsf{ch}}{}^{\ddagger}$ on various Church numerals has a consistent 200% speedup over using TDPE directly to perform the same specialization. This substantially higher speedup over the Tiny experiment is also due to the lack of computational effect in the case of Church numerals, which enables us to specialize Pure TDPE instead of Full TDPE. Since the control operators shift and reset used in Full TDPE to handle possible side effects and sum types cannot be specialized away, they remain to be an obstacle to a higher speedup in specializing Full TDPE.

7 Conclusions and Issues

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. Our experience with the second Futamura projection gave additional evidence that binding-time annotation for TDPE is relatively easy when compared with traditional offline partial evaluators.

When using the hand-written cogen approach, researchers often implement the generating extension starting from a partial evaluator, thereby reducing the implementation and the accompanying the correctness of the cogen to those of the partial evaluator [3, 24, 25]. The reduction itself, however, is generally quite non-trivial. In using the second Futamura projection to produce generating extensions, we have further reused TDPE for the reductions of the implementation and the correctness.

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, if possible at all, to implement the third Futamura projection directly in ML. Even if it could be done, our experience with the second Futamura projection suggests that only an insignificant speedup could 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 Section 5 clearly are 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 to handle every language feature that is used statically. Even so, TDPE would still retain a distinct flavor when compared to traditional partial evaluation techniques: only those constructs not handled automatically by TDPE need to be binding-time annotated. This simplifies the binding-time analysis considerably and often makes binding-time improvements unnecessary, which was one of the original motivations of TDPE.

Acknowledgments

We are indebted to Olivier Danvy, Andrzej Filinski and Morten Rhiger for fruitful discussions on the subject. At an early stage both Olivier Danvy and Morten Rhiger [23] independently implemented a similar version of the second Futamura projection, thus providing further stimulation for our work. Andrzej Filinski's formal treatment of TDPE proved to be invaluable for understanding the second Futamura projection for TDPE. Eijiro Sumii pointed out how the monomorphizing transformations can be improved (see Section 4.4). Thanks are also due to Daniel Damian, Olivier Danvy, Andrzej Filinski, Lasse R. Nielsen and the anonymous referees for their helpful comments.

References

[1] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Albert R. Meyer, editor, *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–213, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[2] 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.

[3] Anders Bondorf and Dirk Dussart. Improving CPS-based partial evaluation: Writing cogen by hand. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report

94/9, University of Melbourne, Australia, pages 1–10, Orlando, Florida, June 1994.

[4] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154, Copenhagen, Denmark, June 1993. ACM Press.

[5] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. In *Partial Evaluation, Proceedings*, volume 1110 of *Lecture Notes in Computer Science*, pages 73–94. Springer-Verlag, 1996.

[6] Olivier Danvy. Type-directed partial evaluation. In *Proceedings of 21st Annual Symposium on Principles of Programming Languages*, pages 242–257. ACM Press, 1996.

[7] Olivier Danvy. A simple solution to type specialization. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of ICALP '98*, number 1443 in Lecture Notes in Computer Science, pages 908–917. Springer, 1998.

[8] 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*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag. Extended version available as BRICS technical report LN-98-3.

[9] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[10] Olivier Danvy and Morten Rhiger. Compiling actions by partial evaluation, revisited. Technical Report BRICS-RS-98-13, BRICS, Department of Computer Science, University of Aarhus, June 1998.

[11] 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. Springer-Verlag. Extended version available as BRICS technical report RS-96-13.

[12] Andrzej Filinski. Representing monads. In Hans-J. Boehm, editor, *Proceedings of the Twenty-First Annual ACM Symposium on Principles*

of Programming Languages, pages 446–457, Portland, Oregon, January 1994. ACM Press.

[13] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *International Conference on Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 378–395, Paris, France, September 1999. Springer-Verlag.

[14] Yoshihito Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):363–397, 1999. Reprinted from Systems · Computers · Controls 2(5), 1971.

[15] Bernd Grobauer and Zhe Yang. Source code for the second Futamura projection for type-directed partial evaluation in ML. Available from `http://www.brics.dk/~grobauer/second_Futamura/sources.tar.gz`.

[16] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7:507–541, 1997. Extended version available as BRICS technical report RS-96-34.

[17] Carsten K. Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, 4^{th} Annual Glasgow Workshop on Functional Programming, Skye, Scotland*, pages 210–218. Glasgow University, 1991.

[18] 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.

[19] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall International, 1993.

[20] 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.

[21] 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.

[22] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Olivier Danvy, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99), Proceedings*, Technical report BRICS-NS-99-1, BRICS, Department of Computer Science, University of Aarhus, pages 25–29, 1999.

[23] Morten Rhiger. Run-time code generation for type-directed partial evaluation. Progress report, BRICS PhD School, University of Aarhus. Available at `http://www.brics.dk/~mrhiger`, 1999.

[24] Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 180–189, Philadelphia, Pennsylvania, May 1996. ACM Press.

[25] Peter Thiemann. Combinators for program generation. *Journal of Functional Programming*, 9(5):483–525, 1999.

[26] Zhe Yang. Encoding types in ML-like languages. In Paul Hudak and Christian Queinnec, editors, *Proceedings of the 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300, Baltimore, Maryland, September 1998. ACM Press. Extended version available as BRICS technical report RS-98-9.

A Compiler Generation for Tiny

A.1 A Binding-Time-Separated Interpreter for Tiny

Paulson's Tiny language [21] is a prototypical imperative language—the BNF of its syntax is given in Figure 18. Figure 19 displays the factorial function coded in Tiny.

```
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 18: 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 19: Factorial function in Tiny

Experiments in type-directed partial evaluation of a Tiny interpreter with respect to a Tiny program [5, 6] used an ML implementation of a Tiny

interpreter (Figure 20): For every syntactic category a meaning function is defined—see Figure 21 for the ML datatype representing Tiny syntax. The meaning of a Tiny program is a function from stores to stores; 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.

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 2.3, the implementation is made as part of a functor which abstracts over all dynamic constants (for example `cond`, `fix` 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-generating functions; as pointed out in Example 3 and made precise in Definition 8, reflection can be used to write code-generating 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
```

A.2 Generating a Compiler for Tiny

As mentioned in Section 5, we derive a compiler for Tiny in three steps:

1. rewrite `tiny_pe` into a 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 Figure 16, thereby creating the GE-instantiation `''meaning''`
3. use the GE-instantiation `''meaning''` to perform the second Futamura projection

The following three 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 4.3.

A.2.1 "Full parameterization"

Following Section 4.5 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 20 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, respectively. Consider for example case distinction over `Expression`; Figure 22 shows the type of the corresponding elimination function.

The resulting implementation is sketched in Figure 23. 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 24. 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 and dynamically—in this case two distinct names (for example `Int_s` and `Int_d`) have to be used.

As mentioned in Section 5, the monotype of every instance of a constant appearing in the interpreter has to be determined. It is these monotypes that have to be declared in the signatures `STATIC` and `DYNAMIC`. Figure 25 shows a portion of signature `STATIC`: the polymorphic type of `caseExpression` (Figure 22) 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.[2] For base types which occur both in signatures `STATIC` and `DYNAMIC`, sharing constraints have to be declared in the interface of functor `tiny_ge` (Figure 24).

Finding the monotypes for the different instantiations of constants appearing in the interpreter can be facilitated by using the type-inference mechanism of ML: we transcribe the output of ML type inference by hand into a type specification. This transcription is straightforward, because the type specifications of TDPE and the output of ML type inference are very much alike.

[2]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.

A.2.2 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 7), perform type-directed partial evaluation by employing the residualizing instantiation (Definition 8), or carry out the second Futamura projection with the GE-instantiation (Definition 15). Section 4.5 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-generating functions. The latter are constructed using the evaluating and the residualizing instantiation of reflection, respectively. Note that 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.

```
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 20: 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 21: Datatype for representing Tiny programs

```
val case_Expression
  : Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a
```

Figure 22: An elimination function for expressions

```
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 23: 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 24: Parameterizing over both static and dynamic constructs

```
⋮
type 'a case_Exp_type
  = Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a

type case_Exp_res_type
  = (Identifier -> Int_s) -> sto -> Int_d

⋮

val case_Expression: case_Exp_res_type case_Exp_type

⋮
```

Figure 25: Excerpts from signature `STATIC`

Recent BRICS Report Series Publications

RS-99-40 **Bernd Grobauer and Zhe Yang.** ***The Second Futamura Projection for Type-Directed Partial Evaluation*****. November 1999. 44 pp. Extended version of an article appearing in Lawall, editor,** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,*** **PEPM '00 Proceedings, 2000, pages 22–32.**

RS-99-39 **Romeo Rizzi.** ***On the Steiner Tree*** $\frac{3}{2}$***-Approximation for Quasi-Bipartite Graphs*****. November 1999. 6 pp.**

RS-99-38 **Romeo Rizzi.** ***Linear Time Recognition of*** P_4***-Indifferent Graphs*****. November 1999. 11 pp.**

RS-99-37 **Tibor Jordán.** ***Constrained Edge-Splitting Problems*****. November 1999. 23 pp. A preliminary version with the title** ***Edge-Splitting Problems with Demands*** **appeared in Cornujols, Burkard and Wöginger, editors,** ***Integer Programming and Combinatorial Optimization: 7th International Conference,*** **IPCO '99 Proceedings, LNCS 1610, 1999, pages 273–288.**

RS-99-36 **Gian Luca Cattani and Glynn Winskel.** ***Presheaf Models for CCS-like Languages*****. November 1999. ii+46 pp.**

RS-99-35 **Tibor Jordán and Zoltán Szigeti.** ***Detachments Preserving Local Edge-Connectivity of Graphs*****. November 1999. 16 pp.**

RS-99-34 **Flemming Friche Rodler.** ***Wavelet Based 3D Compression for Very Large Volume Data Supporting Fast Random Access*****. October 1999. 36 pp. Extended version of paper appearing in** ***7th Pacific Conference on Computer Graphics and Applications,*** **PG '99 Proceedings, 1999, pages 108–117.**

RS-99-33 **Luca Aceto, Zoltán Ésik, and Anna Ingólfsdóttir.** ***The Max-Plus Algebra of the Natural Numbers has no Finite Equational Basis*****. October 1999. 25 pp. To appear in** ***Theoretical Computer Science*****.**

RS-99-32 **Luca Aceto and François Laroussinie.** ***Is your Model Checker on Time? — On the Complexity of Model Checking for Timed Modal Logics*****. October 1999. 11 pp. Appears in Kutyłowski, Pacholski and Wierzbicki, editors,** ***Mathematical Foundations of Computer Science: 24th International Symposium,*** **MFCS '99 Proceedings, LNCS 1672, 1999, pages 125–136.**