Basic Research in Computer Science

BRICS RS-00-44 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-00-44**

ISSN 0909-0878 **December 2000**

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

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

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

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/00/44/`

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

December, 2000

Abstract

A generating extension of a program specializes the program with respect to part of the input. Applying a partial evaluator to the program trivially yields a generating extension, but specializing the partial evaluator with respect to the program often yields a more efficient one. This specialization can be carried out by the partial evaluator itself; in this case, the process is known as the second Futamura projection.

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 (http://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; supported by BRICS and NSF CCR-9970909.

Contents

List of Figures

1 Introduction

1.1 Background

General notions Given a general program $\mathsf{p} : \sigma_S \times \sigma_D \to \sigma_R$ and a fixed *static* input $\mathsf{s} : \sigma_S$, partial evaluation (a.k.a. program specialization) yields a specialized program $\mathsf{p_s} : \sigma_D \to \sigma_R$. When this specialized program $\mathsf{p_s}$ is applied to an arbitrary *dynamic* input $\mathsf{d} : \sigma_D$, it produces the same result as the original program applied to the complete input (s, d), i.e., $[\![\mathsf{p_s d}]\!] = [\![\mathsf{p}(\mathsf{s}, \mathsf{d})]\!]$ (where $[\![\cdot]\!]$ maps a piece of program text to its denotation. In this article, metavariables in *slanted serif* font, such as p, s, and d stand for program terms. Meanwhile, variables in *italic* font, such as x and y, are normal variables in the subject program). Often, some computation in program p can be carried out independently of the dynamic input d, and hence the specialized program $\mathsf{p_s}$ is more efficient than the general program p. In general, specialization is carried out by performing the computation in the source program p that depends only on the static input s, and generating program code for the remaining computation (called residual code). A partial evaluator PE is a program that performs partial evaluation automatically, i.e., if PE terminates on p and s then

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

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

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

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

Self-application The above formulation was first given in 1971 by Futamura [17] in the context of compiler generation—the generating extension of an interpreter is a compiler—and is called the *second Futamura projection*. Turning it into practice, however, proved to be much more difficult than what its seeming simplicity suggests; it was not until 1985 that Jones's group implemented Mix [23], the very first effective self-applicable partial evaluator. They identified the reason for previous failures: The decision whether to carry out computation or to generate residual code generally depends on the static input s, which is not available during self-application; so the specialized partial evaluator still bears this overhead of decision-making. They solved the problem by taking the decision *offline*, i.e., the source program p is pre-annotated with binding-time annotations that solely determine the decisions of the partial evaluator. In the simplest form, a binding time is either static, which indicates computation carried out at partial evaluation time (hence called static computation), or dynamic, which indicates code-generation for the specialized program.

Subsequently, a number of self-applicable partial evaluators have been implemented, e.g., Similix [3], but most of them are for untyped languages. For typed languages, the so-called *type specialization* problem arises [21]: Generating extensions produced using self application often retain a universal data type and the associated tagging/untagging operations as a source of overhead. The universal data type is necessary for representing static values in the partial evaluator, just as it is necesssary for representing values in a standard evaluator. This is unsurprising, because a partial evaluator acts as a standard evaluator when all input is static.

Partly because of this, in the 1990's, the practice shifted towards hand-written generating-extension generators [2, 20]; this is also known as the *cogen approach*. Conceptually, a generating-extension generator is a staged partial evaluator, just as a compiler is a staged interpreter. Ideally, producing a generating extension through self-application of the partial evaluation

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

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

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, normalizing the application of the K-combinator $K = \lambda x.\lambda y.x$ to itself yields $\lambda y.\lambda x.\lambda y'.x$.

Type-directed partial evaluation (TDPE), due to Danvy [5], realizes the above idea using a technique that turned out to be Berger and Schwichtenberg's *Normalization by Evaluation* (NbE) [1, 8]. 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* $\texttt{Exp} \to \texttt{Exp} \to \texttt{Exp} \to \texttt{Exp}$*. We want to extract a* β*-normal form from its meaning.*

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

TDPE is different from a traditional, syntax-directed offline partial evaluator [22] in several respects:

Binding-Time Annotation In traditional partial evaluation, all subexpressions require binding-time annotations. It is unrealistic for the user to annotate the program fully by hand. Fortunately, these annotations are usually computed by an automatic binding-time analyzer,

while the user only needs to provide binding-time annotations on input arguments. On the other hand, since the user does not have direct control over the binding-time annotations, he often needs to know how the binding-time analyzer works and to tune the program in order to ensure termination and a good specialization result.

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

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

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

Flexibility Traditional partial evaluators need to handle all constructs used in a subject program, evaluating the static constructs and generating code for the dynamic ones. In contrast, TDPE uses the underlying evaluator for the static part. Therefore, all language constructs can be used in the static part of a subject program. However, we shall see that this flexibility is lost when self-applying TDPE.

These differences have contributed to the successful application of TDPE in various contexts, e.g., to perform semantics-based compilation [12]. An introductory account, as well as a survey of various treatments concerning NbE, can be found in Danvy's lecture notes [7].

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.

- TDPE extracts the normal form of a term according to a type that can be assigned to the term. This type is supplied in some form of encoding as an argument to TDPE. We can use self-application to specialize TDPE with respect to a particular type; the result helps one to visualize a particular instance of TDPE. This form of self-application was

carried out by Danvy in his original article on TDPE [5]. However, it does not correspond to the second Futamura projection, because no further specialization with respect to a particular subject program is carried out.

- The aforementioned form of self-application [5] was carried out in the untyped language Scheme. It is not immediately clear whether self-application can be achieved in a language with Hindley-Milner type system, such as ML [25]: Whereas TDPE can be implemented in Scheme as a function that takes a type encoding as its first argument, this strategy is impossible in ML, because such a function would require a dependent type. Indeed, the ML implementation of TDPE uses the technique of type encodings [32]: For every type, a particular TDPE program is constructed. As a consquence, the TDPE algorithm to be specialized is not fixed.

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

TDPE also poses some technical problems for self-application. For example, TDPE treats monomorphically typed programs, but the standard call-by-value TDPE algorithm itself uses the 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 language with Hindley-Milner type system. To generate efficient generating extensions, such as compilers, we reformulate the second Futamura projection in a way that is suitable for TDPE.

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

Since TDPE is both the tool and the subject program involved in self-application, we 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 provides an index of notation (Appendix A) and gives further technical details in the generation of a Tiny compiler (Appendix B). The complete source code of the development presented in this article is available online [18].

2 TDPE in a nutshell

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

2.1 Pure TDPE in ML

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

NbE works by extracting the normal form of a λ-term from its meaning, by regarding the term as a higher-order code-manipulation function. The extraction functions are type-indexed coercion functions coded in the object language. To carry out partial evaluation based on NbE, TDPE thus needs to prepare a code-manipulation version of the subject λ-term. Such a λ-term, in general, could contain constant functions that cannot be statically evaluated; these constants have to be replaced with code-manipulating functions.

Pure simply-typed λ-terms We first consider TDPE only for pure simply-typed λ-terms. We use the type `Exp` in Figure 1 to represent code (as it is used in Example 1 on page 5). 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 that correspond to pure λ-terms with type τ of the form $\tau ::= \bullet \mid \tau_1 \to \tau_2$, where '$\bullet$' denotes a base type. 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). Reification is the function that extracts a normal form from the value of a code-manipulation function, using reflection as an auxiliary function. We explain reification and reflection through the following examples.

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

Figure 2: Reification and reflection

Example 2. *We revisit the normalization of* KK *from Example 1 on page 5. For the type* $\bullet \to \bullet \to \bullet \to \bullet$ *the equations given in Figure 2 define reification as*

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

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

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 variable, since the types do not match: $\overline{\tau_1} \neq \texttt{Exp}$. This is where the *reflection* function $\uparrow_\tau : \texttt{Exp} \to \overline{\tau}$ 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 the application of* e *to these term representations. It is used, e.g., when reifying the term* $\lambda x.\lambda y.xyyy$ *with* $\downarrow^{(\bullet \to \bullet \to \bullet \to \bullet) \to \bullet \to \bullet}$.

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

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

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

where $\underline{\text{“mult”}}$ *(=* `VAR` *“mult”) is the code representation of a constant with name mult. Now applying the reification function* $\downarrow^{\bullet\to\bullet}$ *to the term*

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

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

Partial evaluation

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

In general, to perform TDPE for a term, one needs to decide for each constant occurrence, whether to use the original constant or a code-generating instantiation of it; appropriate lifting functions have to be inserted where necessary. The result must type-check, and its partial application to the static input must represent a code-manipulation function (i.e., its type is built up from only the base type `Exp`), so that we can apply the reification function.

This process of classification corresponds to a binding-time annotation phase, as will be made precise in the framework of a two-level language

(Section 2.3). Basically, a source term is turned into a well-formed two-level term by marking constants as static or dynamic, inserting lifting functions where needed. Because only constant occurrences have to be annotated, this can, in practice, be done by hand. Given an annotated term t_{ann}, we call the corresponding code-manipulation function its *residualizing instantiation* $\overline{t_{\mathsf{ann}}}$. It arises from t_{ann} by instantiating each dynamic constant c with its code-generating version c_{r}, each static constant with itself, and each lifting function with the appropriate coercion function into `Exp`. If t is of type σ, then its normal form can be calculated by reifying $\overline{t_{\mathsf{ann}}}$ at type σ (remember that reification only distinguishes a type's form—all base types are treated equally as '•'):

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

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

- binding-time annotate p and s as p_{ann} and s_{ann}, respectively;[1] the term $\lambda x.\overline{p_{\mathsf{ann}}}(\overline{s_{\mathsf{ann}}}, x)$ must be a code-manipulation function of type $\overline{\sigma_D \to \sigma_R}$ (recall that $\overline{\tau}$ arises from τ by instantiating each base type with `Exp`).
- carry out partial evaluation by reifying the above term at type $\sigma_D \to \sigma_R$:

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

Example 5. *Consider the function*

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

Suppose we want to specialize `height` *to a static input* a:`real`. *It is easy to see that the computation of* `sin` *can be carried out statically, but the computation of* `mult` *cannot—*`mult` *is a dynamic constant. This analysis results in a two-level term* $\texttt{height}_{\mathsf{ann}}$, *in which* `sin` *is marked as static,* `mult` *as dynamic, and a lifting function has been inserted to make the static result of applying* `sin` *to a dynamic. The residualizing instantiation of* $\texttt{height}_{\mathsf{ann}}$ *instantiates* `sin` *with the standard sine function, the lifting function with a coercion function from* `real` *into* `Exp`, *and* `mult` *with a code-generating version as introduced in Example 4 on the page before:*

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

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

[1] That the static input also needs to be binding-time annotated may at first seem strange. This is natural, however, because TDPE also accepts higher-order values as static input. For a static input of base type, the binding-time annotation is trivial.

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

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 [6, 32]; see also Rhiger's derivation [28]. 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 on the following page shows, we specify these combinators in a signature called `NBE`. Their implementation as the functor `makePureNbE`—parameterized over two structures of respective signatures `EXP` (term representation) and `GENSYM` (name generation for variables)—is given in Figure 4 on page 14. The implementation is a direct transcription from the formulation in Section 2.1.

Example 6. *We implement an* `NBE`*-structure* `PureNbE` *by applying the functor* `makePureNbE` *(Figure 4 on page 14); this provides us with combinators* `-->` *and* `a'` *and functions* `reify` *and* `reflect`*. Normalization of* KK *(see Example 1 on page 5 and Example 2 on page 9) is carried out as follows:*

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

After evaluation, the variable `KK_norm` *is bound to a term representation of the normal form of* KK*.*

Encoding two-level terms through functors

As mentioned earlier, the input to TDPE needs to be binding-time annotated, i.e., the input is a two-level term. The ML module system makes it possible to encode a two-level term p in a convenient way: Define p inside a functor `p_pe(structure D: DYNAMIC) = ...` which parameterizes over all dynamic types, dynamic constants and lifting functions. By instantiating `D` with an appropriate structure, one can create either the evaluating instantiation $\widetilde{p}$ or the residualizing instantiation $\overline{p}$.

```
signature NBE =                                  (* normalization by evaluation *)
  sig
    type Exp
    type 'a rr                                   (* (↓^τ, ↑_τ) : τ rr *)

    val a' : Exp rr                              (* τ = • *)
    val --> : 'a rr * 'b rr -> ('a -> 'b) rr     (* τ = τ1 → τ2 *)
      ⋮
    val reify: 'a rr -> 'a -> Exp                (* ↓^τ *)
    val reflect: 'a rr -> Exp -> 'a              (* ↑_τ *)
  end

signature EXP =                                  (* term representation *)
  sig
    type Exp
    type Var

    val VAR: Var -> Exp
    val LAM: Var * Exp -> Exp
    val APP: Exp * Exp -> Exp
     ⋮
  end

signature GENSYM =                               (* name generation *)
  sig
    type Var
    val new: unit -> Var                         (* make a new name *)
    val init: unit -> unit                       (* reset name counter *)
  end;
```

Figure 3: NbE in ML, signatures

Example 7. *In Example 5 on page 11 we sketched how the function* `height` *can be partially evaluated with respect to its first argument. Figure 5 on page 15 shows how to provide both evaluating and residualizing instantiation in ML using functors. The two-level term* $\texttt{height}_{\texttt{ann}}$ *is encoded as a functor* `height_pe(structure D:DYNAMIC)` *that is parameterized over the dynamic type* `Real`*, the dynamic constant* `mult`*, and the lifting function* `lift_real` *in* $\texttt{height}_{\texttt{ann}}$.

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

```
functor makePureNbE(structure G: GENSYM
                    structure E: EXP
                    sharing type E.Var = G.Var): NBE =
  struct
    type Exp = E.Exp
    datatype 'a rr = RR of ('a -> Exp) * (Exp -> 'a)
                                                  (* (↓^τ, ↑_τ) : τ rr *)

    infixr 5 -->
    val a' = RR(fn e => e, fn e => e)                    (* τ = • *)
    fun RR (reif1, refl1) --> RR(reif2, refl2)     (* τ = τ1 → τ2 *)
      = RR (fn f =>
              let val x = G.new ()
                in E.LAM (x, reif2 (f (refl1 (E.VAR x))))
              end,
            fn e =>
              fn v => refl2 (E.APP (e, reif1 v)))

    fun reify (RR (reif, refl)) v                        (* ↓^τ *)
      = (G.init (); reif v)
    fun reflect (RR (reif, refl)) e                      (* ↑_τ *)
      = refl e
  end
```

Figure 4: Pure NbE in ML, implementation

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.

Sum types and call-by-value languages can be handled by manipulating the code-generation context in the reflection function. This has been achieved by using the control operators shift and reset [9, 14]. Section 4.5 describes in more detail the treatment of sum types and call-by-value languages in TDPE.

Figure 6 on page 16 displays the signature `CTRL` of control operators and the skeleton of a functor `makeFullNbE` that is used to implement Full TDPE—an implementation can be found in Danvy's lecture notes [7]. The relevance of Full TDPE in this article is that (1) it is the partial evaluator that one would use for specializing realistic programs; and (2) in particular, it handles all features used in its own implementation, including side effects and control effects. Hence in principle self-application should be possible.

```
signature DYNAMIC =        (* Signature of dynamic types and constants *)
  sig
    type Real

    val mult: Real -> Real -> Real
    val lift_real: real -> Real
  end

                                  (* The functor encodes a two-level term *)
functor height_pe(structure D: DYNAMIC) =
 struct
   fun height a z = D.mult (D.lift_real (sin a)) z
 end


structure EDynamic: DYNAMIC =                          (* Defining ~· *)
  struct
    type Real = real
    fun mult x y  =  x * y
    fun lift_real r = r
  end

structure RDynamic: DYNAMIC =                          (* Defining ¯· *)
  struct
    local
      open EExp PureNbE
      infixr 5 -->
    in
      type Real = Exp
      val mult = reflect (a' --> a' --> a') (VAR "mult")
      fun lift_real r  = LIT_REAL r
    end
  end


structure Eheight = height_pe (structure D = EDynamic);
                                                    (* height~ann *)
structure Rheight = height_pe (structure D = RDynamic);
                                                    (* ¯height¯ann *)
```

Figure 5: Instantiation via functors

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

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

Signatures `GENSYM`, `EXP`, and `NBE` are defined in Figure 3 on page 13.

Figure 6: Full NbE in ML.

2.3 A general account of TDPE

The introduction to TDPE given in Section 2.1 is concerned with providing intuition rather than formal detail; in the following, we describe Filinski's formalization of TDPE [16], which gives a precise definition to the concepts that were introduced only informally before. This formal account is rather technical and may be skipped on first reading: When developing self-application for TDPE in Section 3, we shall start with an intuitive account that can be understood without having read the following material. Nevertheless, the details of the development turn out to be rather intricate, so an informal account alone is not satisfactory. In Section 3.2 we draw upon the formal account of TDPE presented here, and derive a formulation of self-application from it.

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".

The denotational semantics of types and terms is determined by an interpretation. An interpretation $\mathcal{I}$ of signature Σ assigns domains to base type names, elements of appropriate domains to literals and constants, and, in the setting of call-by-value languages with effects, also monads to various effects. The interpretation $\mathcal{I}$ extends canonically to the meaning $[\![\sigma]\!]^{\mathcal{I}}$ of

every type σ and the meaning $[\![t]\!]^{\mathcal{I}}$ of every term $t : \sigma$ in the language; we write $[\![t]\!]^{\mathcal{I}}$ for closed terms t, which denote elements in the domain $[\![\sigma]\!]^{\mathcal{I}}$.

The syntactic counterpart of the notion of an interpretation is that of an instantiation, which compositionally maps syntactic phrases in a language L to syntactic phrases in (usually) another language L'. The following definition of instantiations uses the notion of substitutions. For a substitution Φ, we write $t\{\Phi\}$ and $\tau\{\Phi\}$ to denote the application of Φ to term t and type τ, respectively.

Definition 8 (Instantiation). *Let L and L' be two languages with signatures Σ and Σ', respectively. An instantiation Φ of Σ-phrases (terms and types) into language L' is a substitution that maps the base types in Σ to Σ'-types, and maps constants $c : \sigma$ to closed Σ'-terms of type $\sigma\{\Phi\}$.*

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

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

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*. The meaning $[\![e]\!]^{\mathcal{I}^2}$ of a term e is 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; we also write $[\![e]\!]^{\mathcal{I}^{\mathfrak{s}},\mathcal{I}^{\mathfrak{d}}}$ for $[\![e]\!]^{\mathcal{I}^2}$. A two-level language is different from a one-level language in that the meaning of terms is parameterized over the dynamic interpretation $\mathcal{I}^{\mathfrak{d}}$. More precisely, it is specified by a pair $(\Sigma^2, \mathcal{I}^{\mathfrak{s}})$ of its signature Σ^2 and a fixed static interpretation $\mathcal{I}^{\mathfrak{s}}$.

A two-level language $PL^2 = (\Sigma^2, \mathcal{I}^{\mathfrak{s}})$ is usually associated with a one-level language $PL = (\Sigma^{PL}, \mathcal{I}^{PL})$:

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. All these constructs are superscripted by $\mathfrak{s}$ in $\Sigma^{\mathfrak{s}}$.

3. The static interpretation $\mathcal{I}^{\mathfrak{s}}$ is the restriction of interpretation $\mathcal{I}^{PL}$ to $\Sigma^{\mathfrak{s}}$.

For clarity, we let metavariable t range over one-level terms, e over two-level terms, σ over one-level types, and τ over two-level types.

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 $[\![\$_{\mathsf{b}}]\!]^{\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = (\lambda x.x) \in [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}} \to [\![\mathsf{b}]\!]^{\mathcal{I}^{PL}}$. A closely related notion is the *evaluating instantiation* of Σ^2-phrases in Σ^{PL}:

Definition 9 (Evaluating Instantiation). *The* evaluating instantiation *of a Σ^2-term $\vdash_{\Sigma^2} e{:}\tau$ in PL is $\vdash_{\Sigma^{PL}} \widetilde{e}{:}\widetilde{\tau}$, given by $\widetilde{e} = e\{\Phi_\sim\}$ 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): $\Phi_\sim(\mathsf{b}^{\mathfrak{s}}) = \Phi_\sim(\mathsf{b}^{\mathfrak{d}}) = \mathsf{b}$, $\Phi_\sim(c^{\mathfrak{s}}) = \Phi_\sim(c^{\mathfrak{d}}) = c$, $\Phi_\sim(\$_{\mathsf{b}}) = \lambda x.x$.*

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

Static normalization Static normalization works on Σ^2-terms of fully dynamic types, i.e., types constructed solely from dynamic base types. A term 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 with terms $\widetilde{e}$ 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 $e' = NF(e)$ 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., $\forall \mathcal{I}^{\mathfrak{d}}.[\![e]\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}} = [\![e']\!]^{\mathcal{I}^{\mathfrak{s}}, \mathcal{I}^{\mathfrak{d}}}$; 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 reduce the static normalization function *NF* for a two-level language PL^2 to evaluation in the ordinary language *PL*. For this to be possible, we assume that language *PL* is equipped with a base type `Exp` for the representation of its own terms (and thus of static normal forms in PL^2), and constants that support name generation and code construction (for example, a lifting function $\mathit{lift}_{\mathsf{b}} : \mathsf{b} \to \mathtt{Exp}$ for every base type b).

Filinski has shown that in the described setting, NbE can be performed with two type-indexed functions $\downarrow^{\tau} : \overline{\tau} \to \mathtt{Exp}$ (reification) and $\uparrow_{\tau} : \mathtt{Exp} \to \overline{\tau}$ (reflection)—here the operation $\overline{\cdot}$ on two-level types corresponds to the

one introduced in Section 2.1 for ML types; a formal definition is given in Definition 10. The function $\downarrow^\tau$ extracts the static normal form of a term $\vdash_{\Sigma^2} e{:}\tau$ from a special *residualizing instantiation* of the term in PL, $\vdash_{\Sigma^{PL}} \overline{e}{:}\overline{\tau}$, and the function $\uparrow_\tau$ is used in both the definition of reification function and the construction of the residualizing instantiation $\overline{e}$.

Definition 10 (Residualizing Instantiation). *The* residualizing instantiation *of a Σ^2-term $\vdash_{\Sigma^2} e : \tau$ in PL is $\vdash_{\Sigma^{PL}} \overline{e} : \overline{\tau}$, given by $\overline{e} = e\{\Phi_{\sqcap}\}$ and $\overline{\tau} = \tau\{\Phi_{\sqcap}\}$, where instantiation $\Phi_{\sqcap}$ is a substitution of Σ^2-constructs into Σ^{PL}-phrases: for base types b, $\Phi_{\sqcap}(\mathsf{b}^{\mathsf{s}}) = \mathsf{b}$, $\Phi_{\sqcap}(\mathsf{b}^{\mathsf{d}}) = \mathtt{Exp}$, for constants c, $\Phi_{\sqcap}(c^{\mathsf{s}}) = c$, $\Phi_{\sqcap}(c^{\mathsf{d}} : \tau) = \uparrow_\tau \underline{\text{“}c\text{”}}$, and for lifting functions over a base type b, $\Phi_{\sqcap}(\$_{\mathsf{b}}) = \mathit{lift}_{\mathsf{b}}$.*

In words, the residualizing instantiation $\overline{\tau}$ of a fully dynamic type τ substitutes all occurrences of dynamic base types in τ with type $\mathtt{Exp}$. Since type τ is fully dynamic, type $\overline{\tau}$ is constructed from type $\mathtt{Exp}$, 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 versions (cf. Example 5 on page 11, where $\mathtt{height}_{\mathsf{ann}}$ is $\lambda\,(a : \mathtt{real}^{\mathsf{s}}).\lambda\,(z : \mathtt{real}^{\mathsf{d}}).\mathtt{mult}^{\mathsf{d}}\ (\$_{\mathtt{real}}(\mathtt{sin}^{\mathsf{s}}\ a))\ z)$.

The function *NF* in NbE is defined by Equation (1) on the next page) in Figure 7 on the following page: It computes the static normal form of term e by evaluating the Σ^{PL}-term $\vdash_{\Sigma^{PL}} \downarrow^\tau \overline{e} : \mathtt{Exp}$ 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 11 (Filinski [16]). *The function NF defined in Equation (1) in Figure 7 on the next page 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 $\forall \mathcal{I}^{\mathsf{d}}.[\![e]\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathsf{d}}} = [\![e']\!]^{\mathcal{I}^{\mathsf{s}},\mathcal{I}^{\mathsf{d}}}$.*

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 11. The details of how Theorem 11 is proved are out of the scope of this article.

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) on the following page) to statically normalize the trivial specialization $\lambda x.p(s, x)$:

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

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 $\vdash_{\Sigma^{PL}} \downarrow^\tau : \overline{\tau} \to \mathtt{Exp}$ is the reification function for type τ.

Binding-time annotation The task is, given $\vdash_{\Sigma^{PL}} t : \sigma$ and binding-time constraints in the form of a two-level type τ whose erasure is σ, to find $\vdash_{\Sigma^2} t_{\mathsf{ann}} : \tau$ that satisfies the constraints and the following equations:

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

Figure 7: A formal recipe for NbE

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 instantiation $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathsf{s}}, \mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}}$ 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: Forming the application of t_{ann} to 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 $\vdash_{\Sigma^2} t_{\mathsf{ann}} \equiv \lambda x.p_{\mathsf{ann}}(s_{\mathsf{ann}}, x) : \tau_D \to \tau_R$. It corresponds to a one-level term $t \equiv \lambda x.p(s, x)$, for which (by compositionality of

the meaning functions) $[\![t_{\mathsf{ann}}]\!]^{\mathcal{I}^{\mathfrak{s}},\mathcal{I}^{\mathfrak{d}}_{\mathsf{ev}}} = [\![t]\!]^{\mathcal{I}^{PL}}$. 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:

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

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, the constructors associated with `Exp`, 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 between *PL* and ML. The associated two-level language PL^2 is constructed from the language *PL* mechanically. As shown in Section 2.2 (Example 7 on page 13), a two-level term can be encoded in ML by using a functor to parameterize over all dynamic types and constants in the term. Instantiating the functor with a structure that defines either the original constants or their code-generating versions yields the evaluating instantiation or the residualizing instantiation, respectively.

3 Formulating self-application

In this section, we present two forms of self-application for TDPE. One uses self-application to generate more efficient reification and reflection functions for a type τ; following Danvy [5], we refer to this form of self-application as *visualization*. The other adapts the second Futamura projection to the setting of TDPE. We first give an intuitive account of how self-application can be achieved, and then derive a precise formulation of self-application, based on the formal account of TDPE presented in Section 2.3.

3.1 An intuitive account of self-application

We start by presenting the intuition behind the two forms of self application, drawing upon the informal account of TDPE in Section 2.1.

Visualization

For a specific type τ, the reification function $\downarrow^{\tau}$ contains one β-redex for each recursive call following the type structure. For example, the direct unfolding of $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$, according to the definition (Figure 2 on page 9), is

$$\lambda f_0.\underline{\lambda x}.(\lambda f_1.\underline{\lambda y}.(\lambda f_2.\underline{\lambda z}.(\lambda e.e)(f_2((\lambda e.e)\underline{z})))(f_1((\lambda e.e)\underline{y})))(f_0((\lambda e.e)\underline{x}))$$

rather than the normalized form presented in Example 2 on page 9. This normalization can be achieved by self-applying TDPE so as to specialize the reification function with respect to a particular type. Danvy has carried out this form of self application in the untyped language Scheme [5]; in the following, we reconstruct it in our setting.

Recall from Section 2 that finding the normal form of a term $t : \sigma$ is achieved by reifying the residualizing instantiation of a binding-time annotated version of t:

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

It thus suffices to find an appropriate binding-time annotated version of the term $\downarrow^{\tau}$. A straightforward analysis of the implementation of NbE (see Figure 3 on page 13 and Figure 4 on page 14), shows that all the base types (`Exp`, `Var`, etc.) and constants (`APP`, `Gensym.init`, etc.[2]) are needed in the code generation phase; hence they all should be classified as dynamic. Therefore, to normalize $\downarrow^{\tau} : \overline{\tau} \to \mathtt{Exp}$, we use a trivial binding-time annotation, notated as $\underline{\langle \cdot \rangle}$, in which every constant is marked as dynamic:

$$NF(\underline{\langle \downarrow^{\tau} \rangle}) = [\![\downarrow^{\tau \to \bullet} \overline{\underline{\langle \downarrow^{\tau} \rangle}}]\!], \tag{3}$$

In order to understand the term $\overline{\underline{\langle \downarrow^{\tau} \rangle}}$, we analyze the composite effect of the residualizing instantiation and trivial binding-time annotation: for a term e, the term $\overline{\underline{\langle e \rangle}}$ is formed from e by substituting all constants with their code-generating counterparts. We write $\Downarrow^{\tau}$ for $\overline{\underline{\langle \downarrow^{\tau} \rangle}}$ and $\Uparrow_{\tau}$ for $\overline{\underline{\langle \uparrow_{\tau} \rangle}}$ for notational conciseness.

Term $\downarrow^{\tau}$ and term $\Downarrow^{\tau}$ are respectively the evaluating instantiation and residualizing instantiation of the same (two-level) term $\underline{\langle \downarrow^{\tau} \rangle}$: that is, $\widetilde{\underline{\langle \downarrow^{\tau} \rangle}} = \downarrow^{\tau}$, and $\overline{\underline{\langle \downarrow^{\tau} \rangle}} = \Downarrow^{\tau}$; term $\uparrow_{\tau}$ and term $\Uparrow_{\tau}$ have an analogous relationship. We will exploit this fact in Section 4.1 to apply the functor-based approach to the reification/reflection combinators themselves, thus providing an implementation of $\Downarrow^{\tau}$ and $\Uparrow_{\tau}$ in ML.

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 reasonable choice for deriving efficient generating extensions—the evaluator for the language in which we use TDPE might not even be written in this language; making such an evaluator explicit in the partial evaluator to be specialized introduces an extra layer of interpretation, which defeats the advantages of TDPE. We thus consider instead the general idea behind the second Futamura projection:

[2] These constants appear, e.g., in the underlined portion of the expanded term $\downarrow^{\bullet \to \bullet \to \bullet \to \bullet}$.

Using partial evaluation to perform the static computations in a 'trivial' generating extension (usually) yields a more efficient generating extension.

Following the informal recipe for performing TDPE given in Section 2, the 'trivial generating extension' $p^\dagger$ of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ is

$$\lambda s.\, TDPE(p, s) : \sigma_S \to \mathtt{Exp} = \lambda\, s.{\downarrow^{\sigma_D \to \sigma_R}}\ \lambda\, d.\overline{p_{\mathsf{ann}}}(s, d)$$

Since the trivial generating extension is itself a term, we can normalize it using TDPE: We reify at type $\sigma_S \to \bullet$ the residualizing instantiation of the (suitably binding-time annotated) trivial generating extension. We can use the trivial binding-time annotation, i.e., to reify $\overline{\langle\, \lambda s.\, TDPE(p, s)\,\rangle}$ —in Section 3.2 we shall explain in detail why this choice is not too conservative. Because $\overline{\langle\ \cdot\ \rangle}$ is a substitution, it distributes over term constructors, and we can move it inside the terms:

$$\overline{\langle\, \lambda s.\, TDPE(p, s)\,\rangle} = \lambda s.\ {\Downarrow^{\sigma_D \to \sigma_R}}\ (\lambda d.\ \overline{\langle\ \overline{p_{\mathsf{ann}}}\ \rangle}\ (s, d)).$$

For concreteness, the reader might find it helpful to consider the example of the height function (Example 5 on page 11): $\overline{p_{\mathsf{ann}}}$ corresponds to $\overline{\mathtt{height}_{\mathsf{ann}}}$, so $\overline{\langle\ \overline{p_{\mathsf{ann}}}\ \rangle}$ is formed by substituting all the constants in $\overline{\mathtt{height}_{\mathsf{ann}}}$ with their code-generating versions. Such constants include `sin`, $\mathit{lift}_{\mathtt{real}}$, and the code-constructing constants appearing in term $\mathtt{mult}_{\mathsf{r}}$ (Example 4 on page 10).

In practice, however, we do not need to first build the residualizing version by hand and then apply the TDPE formulation. Instead, we show that we can characterize $\overline{\langle\ \overline{e}\ \rangle}$ in terms of the original two-level term e itself, thus enabling a functor-based approach: We write $\overline{\overline{e}}$ for $\overline{\langle\ \overline{e}\ \rangle}$ and call it the *GE-instantiation* of term e, where "GE" stands for *generating extension*. A precise definition of the GE-instantiation is derived formally in Section 3.2 (Definition 17 on page 26). Basically, $\overline{\overline{e}}$ instantiates all static constants and lifting functions in e with their code-generating version and all dynamic constants with versions that generate "code-generating" code. In other words, static constants and lifting functions give rise to code that is executed when applying the generating extension, whereas dynamic constants give rise to code that has to appear in the result of applying the generating extension.

All in all, the generating extension $p^\ddagger$ of a program $p : \sigma_S \times \sigma_D \to \sigma_R$ can be calculated as

$$p^\ddagger = [\![{\downarrow^{\sigma_S \to \bullet}}\ (\lambda s.\ {\Downarrow^{\sigma_D \to \sigma_R}}\ (\lambda d.\overline{\overline{p_{\mathsf{ann}}}}\ (s, d)))]\!]. \tag{4}$$

3.2 A derivation of self-application

In Section 3.1 we gave an intuitive account of how self-application can be achieved for TDPE. Using the formalization of TDPE presented in Section 2.3 we now derive both forms of self-application; correctness thus follows from the correctness of TDPE.

Visualization

We formally derive visualization (Section 3.1), using the "recipe" outlined in Figure 7 on page 20. First, we need a formal definition of the trivial binding-time annotation $\langle\underline{\ \cdot\ }\rangle$ in terms of the two-level language:

Definition 12 (Trivial Binding-Time Annotation). *The* trivial binding-time annotation *of a Σ^{PL}-term $\vdash_{\Sigma^{PL}} \mathsf{t} : \sigma$ is a PL^2-term $\vdash_{\Sigma^2} \langle\underline{\mathsf{t}}\rangle : \langle\underline{\sigma}\rangle$, given by $\langle\underline{\mathsf{t}}\rangle = \mathsf{t}\{\Phi_{\langle\rangle}\}$ and $\langle\underline{\sigma}\rangle = \sigma\{\Phi_{\langle\rangle}\}$, where the instantiation $\Phi_{\langle\rangle}$ is a substitution of Σ^{PL}-constructs into Σ^2-phrases: $\Phi_{\langle\rangle}(\mathsf{b}) = \mathsf{b}^{\mathfrak{d}}$, $\Phi_{\langle\rangle}(\ell : \mathsf{b}) = \$_{\mathsf{b}}\ell^{\mathfrak{s}}$ (ℓ is a literal), $\Phi_{\langle\rangle}(c) = c^{\mathfrak{d}}$ (c is not a literal).*

Lemma 13 (Properties of $\langle\underline{\ \cdot\ }\rangle$). *For a Σ^{PL}-term $\vdash_{\Sigma^{PL}} \mathsf{t} : \sigma$, the following properties hold:*

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

A simple derivation using properties (3) and (4) in Lemma 13, together with the fact that $\langle\underline{\ \cdot\ }\rangle$ and $\overline{\cdot}$ distribute over all type and term constructors, yields the formulation of self-application given in Equation (3) on page 22:

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

The following corollary follows immediately from Theorem 11 on page 19 and property (1) of Lemma 13.

Corollary 14. *If $\mathsf{e}_{\tau} = NF(\langle\underline{\downarrow^{\tau}}\rangle)$, then its one-level representation $\widetilde{\mathsf{e}_{\tau}}$ is free of β-redexes and is semantically equivalent to $\downarrow^{\tau}$:*

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

The self-application carried out by Danvy in the setting of Scheme [5] is quite similar; his treatment explicitly λ-abstracts over the constants occurring in $\downarrow^{\tau}$, which, by the TDPE algorithm, would be reflected according to their types. This reflection also appears in our formulation: For any constant $c : \sigma$ appearing in $\downarrow^{\tau}$, we have $\overline{\langle\underline{c}\rangle} = \overline{c^{\mathfrak{d}}} = \uparrow_{\langle\underline{\sigma}\rangle} \underline{\text{"}c\text{"}}$. Consequently, our result coincides with Danvy's.

Adapted second Futamura projection

We repeat the development from Section 3.1 in a formal way. We begin by rederiving the trivial generating extension, this time from Equation (2) on page 19: 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.

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

Corollary 15 (Trivial Generating Extension). *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 7 on page 20 to specialize it into a more efficient generating extension. We first need to binding-time annotate the term $\mathsf{p}^\dagger$. For the subterm $\downarrow^{\tau_D \to \tau_R}$, the analysis in Section 3.1 shows that we should take the trivial binding-time annotation. For the subterm $\overline{\mathsf{p}}$, the following analysis shows that it is not too conservative to take the trivial binding-time annotation as well. Since $\overline{\cdot} = \Phi_{\sqcap}$ 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 $\Phi_{\sqcap}$. If c' appears inside $\Phi_{\sqcap}(c^{\mathsf{d}}) = \uparrow_\tau \underline{\text{“}c\text{”}}$ (where c' could be a code-constructor such as `LAM`, `APP` appearing in term $\uparrow_\tau$) , or $\Phi_{\sqcap}(\$_{\mathsf{b}}) = \mathit{lift}_{\mathsf{b}}$, then c' is needed in the code generation phase, and hence it should be classified as dynamic. If c' appears inside $\Phi_{\sqcap}(c^{\mathsf{s}}) = c$, 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.

Taking the trivial binding time annotation of the trivial generating extension $\mathsf{p}^\dagger$, we then proceed with Equation (1) on page 20 to generate a more efficient generating extension.

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

Expressing $\overline{\langle \overline{\mathsf{p}} \rangle}$ as $\overline{\overline{\mathsf{p}}}$, and $\overline{\langle \downarrow^{\tau_D \to \tau_R} \rangle}$ as $\Downarrow^{\tau_D \to \tau_R}$, we have

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

as originally given in Equation (4) on page 23.

The generation of $p^\ddagger$ always terminates, even though, in general, the normalization function *NF* may diverge. Recall that the trivial binding-time annotation used in the preceding computation of $p^\ddagger$ marks all constants, including all fixed-point operators, as dynamic. Divergence, however, can only happen when the two-level program contains static fixed-point operators.

The correctness of the second Futamura projection follows from Corollary 15 on the page before and Theorem 11 on page 19.

Corollary 16 (Efficient Generating Extension). *Program* $\widetilde{p^\ddagger}$ *(the one-level form of the static normal form* $p^\ddagger$*) is a generating extension of* p *which is free of* β*-redexes.*

Proof. By Theorem 11 on page 19 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 15 on the preceding page. □

Now let us examine how the term $\ulcorner\overline{p}\urcorner$ is formed. Note that $\ulcorner\overline{p}\urcorner = \overline{\langle \overline{p} \rangle} = ((p\{\Phi_{\sqcap}\})\{\Phi_{\langle\rangle}\})\{\Phi_{\sqcap}\} = p\{\Phi_{\sqcap} \circ \Phi_{\langle\rangle} \circ \Phi_{\sqcap}\}$; 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 17 (GE-instantiation). *The* GE-instantiation *of a* Σ^2*-term* $\vdash_{\Sigma^2} e : \tau$ *in PL is* $\vdash_{\Sigma^{PL}} \ulcorner\overline{e}\urcorner : \ulcorner\overline{\tau}\urcorner$ *given by* $\ulcorner\overline{e}\urcorner = e\{\Phi_{\mathtt{m}}\}$ *and* $\ulcorner\overline{\tau}\urcorner = \tau\{\Phi_{\mathtt{m}}\}$*, where instantiation* $\Phi_{\mathtt{m}}$ *is a substitution of* Σ^2*-constructs into* Σ^{PL}*-phrases:*

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

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 4.3, we show how to extend the instantiation-through-functor approach to cover GE-instantiation as well.

It is instructive to compare the formulation of the second Futamura projection with the formulation of TDPE (Equation (2) on page 19). The crucial common feature is that the subject program p is only instantiated, i.e., only the constants are substituted in the program; this feature makes them amenable to a functor-based treatment and frees them from an explicit interpreter. For TDPE, however, static constants are instantiated with their

standard instantiation, which makes it possible to use built-in constructs (such as case expressions) in the "static parts" of a program. This is not the case for the second Futamura projection, which causes some inconvenience when applying the second Futamura projection, as we shall see in Section 5.

4 The implementation

In this section we treat various issues arising when implementing the abstract formulation of Section 3 in ML. We start with the implementation of the key components for self application, namely the functions $\Downarrow$ and $\Uparrow$, and the GE-instantiation. We then turn to two technical issues. First, we show how to specify the input, especially the types, for the self-application. Second, we show how to modify the full TDPE algorithm, which uses polymorphically typed control operators, such that it is amenable to the TDPE algorithm itself, i.e., amenable to self-application.

4.1 Residualizing instantiation of the combinators

In Section 3.1 we remarked that the terms $\downarrow^\tau$ and $\Downarrow^\tau$ are respectively the evaluating instantiation and the residualizing instantiation of the same two-level term $\langle\underline{\downarrow^\tau}\rangle$. We can again use the ML module system to conveniently implement both instantiations. Recall that we formulated reification and reflection as type-indexed functions, and we implemented them not as a monolithic program, but as a group of combinators, one for each type constructor. These combinators can be plugged together following the structure of a type τ to construct a type encoding as a reification-reflection pair $(\downarrow^\tau, \uparrow_\tau)$. To binding-time annotate $(\downarrow^\tau, \uparrow_\tau)$ as $(\langle\underline{\downarrow^\tau}\rangle, \langle\underline{\uparrow_\tau}\rangle)$, it suffices to parameterize all the combinators over the constants they use: As already mentioned before, because $\langle\underline{\ \cdot\ }\rangle$ is a substitution, it distributes over all constructs in a term, marking all the types and constants as dynamic. These combinators, when instantiated with either an evaluating or a residualizing instantiation, can be combined according to a type τ to yield either $(\downarrow^\tau, \uparrow_\tau)$ or $(\Downarrow^\tau, \Uparrow_\tau)$.

We can directly use the functors `makePureNbE` (Figure 4 on page 14) and `makeFullNbE` (Figure 6 on page 16) to produce the instantiations, because these functors are 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 `EFullNbE` of NbE is produced by applying the functor `makeFullNbE` to the standard evaluating structures `EExp`, `EGensym` and `ECtrl` of the signatures `EXP`, `GENSYM` and `CTRL`, respectively (Figure 8 on the following page—we show the implementations of structures `EExp` and `EGensym`; for structure `ECtrl`, we use Filinski's implementation [14]). Residualizing instantiations `RFullNbE` of Full NbE and `RPureNbE` of Pure NbE result from applying the

```
structure EExp                                  (* Evaluating Inst. ·~ on EXP *)
= struct
    type Var = string
    datatype Exp =
        VAR of string                                              (* v *)
      | LAM of string * Exp                                        (* λx.e *)
      | APP of Exp * Exp                                           (* e1 @ e2 *)
      | PAIR of Exp * Exp                                          (* (e1,e2) *)
      | PFST of Exp                                                (* fst *)
      | PSND of Exp                                                (* snd *)
      | LIT_REAL of real                                           (* $real *)
  end

structure EGensym                          (* Evaluating Inst. ·~ on GENSYM *)
= struct
    type Var = string

    local val n = ref 0
    in fun new () = (n := !n + 1;                    (* make a new name *)
                     "x" ^ Int.toString (!n))
       fun init () = n := 0                          (* reset name counter *)
    end
  end;

(* Evaluating Instantiation *)
structure EFullNbE = makeFullNbE (structure G = EGensym
                                  structure E = EExp
                                  structure C = ECtrl): NBE
```

Figure 8: Evaluating Instantiation of NbE

functors `makePureNbE` and `makePureNbE`, respectively, to appropriate residualizing structures `RGensym`, `RExp`, and `RCtrl` (Figure 9 on the next page).

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 the definition of reflection (see also Example 3 on page 9).

With the residualizing instantiation of reification and reflection at our disposal, we now can perform visualization by following Equation (3) on page 22.

Example 18. *We show the visualization of* $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$ *(cf. Example 2 on page 9) for Pure NbE. Following Equation (3) on page 22, we have to compute* $\downarrow^{(\bullet\to\bullet\to\bullet\to\bullet)\to\bullet}(\Downarrow^{\bullet\to\bullet\to\bullet\to\bullet})$*. This is done in Figure 10 on page 30; it is not difficult to see that the result matches the execution of the term*

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

                                                  (* VAR v  =  VAR @ v *)
  fun VAR v = EExp.APP (EExp.VAR "VAR", v)

                                              (* LAM (v,e)  =  LAM @ (v,e) *)
  fun LAM (v, e) = EExp.APP (EExp.VAR "LAM",
                             EExp.PAIR (v, e))
                                               (* APP (s,t)  =  APP @ (s,t) *)
  fun APP (s, t) = EExp.APP (EExp.VAR "APP",
                             EExp.PAIR (s, t))
  ⋮
end

⋮

                                        (* Residualizing Instantiations *)
structure RFullNbE = makeFullNbE (structure G = RGensym
                                  structure E = RExp
                                  structure C = RCtrl): NBE

structure RPureNbE = makePureNbE (structure G = RGensym
                                  structure E = RExp): NBE
```

Figure 9: Residualizing Instantiation of NbE

`reify (a' --> a' --> a' --> a')` *(see Figure 4 on page 14). 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 numeral 0_{ch}, successor s_{ch}, and the addition function $+_{\mathsf{ch}}$ in Figure 11 on page 31 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 $\overline{\ulcorner +_{\mathsf{ch}} \urcorner} = \overline{+_{\mathsf{ch}}} = +_{\mathsf{ch}}$. Following Equation (4) on page 23, we can compute an efficient generating extension $+_{\mathsf{ch}}{}^{\ddagger}$, as shown in Figure 11 on page 31.

```
local open EFullNbE
  infixr 5 -->
  val Ereify_aaaa_a
    = reify ((a'-->a'-->a'-->a') --> a')      (* ↓^(•→•→•→•)→• *)
  open RPureNbE
  infixr 5 -->
  val Rreify_aaaa = reify (a'-->a'-->a'-->a')      (* ⇓^•→•→•→• *)
in val nf = Ereify_aaaa_a (Rreify_aaaa) end
```

The (pretty-printed) result `nf` is:

$\lambda x_1.$ **let** $r_2 = \mathsf{init}()$ $r_3 = \mathsf{new}()$ $r_4 = \mathsf{new}()$ $r_5 = \mathsf{new}()$
in
$\underline{\lambda\, r_3}.\underline{\lambda\, r_4}.\underline{\lambda\, r_5}.x_1\ \underline{r_3}\ \underline{r_4}\ \underline{r_5}$
end

Figure 10: Visualizing $\downarrow^{\bullet\to\bullet\to\bullet\to\bullet}$

4.3 The GE-instantiation

We generalize the technique of encoding a two-level term p in ML presented at the end of Section 2.2: We code p inside a functor

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

that parameterizes over both static and dynamic constants. With suitable instantiations of the structures `S` and `D`, one thus can create not only the evaluation instantiation $\widetilde{p}$ and the residualizing instantiation $\ulcorner p \urcorner$, but also the GE-instantiation $\ulcorner\!\ulcorner p \urcorner\!\urcorner$. The instantiation table displayed in Table 1 summarizes how to write the components of the three kinds of instantiation functors for `S` and `D`. The table follows easily from the formal definitions of $\widetilde{\cdot}$, $\ulcorner\cdot\urcorner$ and $\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$ via $\Phi_\sim$ (Definition 9 on page 18), $\Phi_{\ulcorner\urcorner}$ (Definition 10 on page 19) and $\Phi_{\ulcorner\!\ulcorner\urcorner\!\urcorner}$ (Definition 17 on page 26), respectively.

		$\widetilde{\cdot}$	$\ulcorner\cdot\urcorner$	$\ulcorner\!\ulcorner\cdot\urcorner\!\urcorner$
S	b^{s}	b	b	Exp
	$c^{\mathsf{s}} : \tau$	c	c	$\uparrow_{\langle\ulcorner\tau\urcorner\rangle}\ \underline{\text{“}c\text{”}}$
D	b^{d}	b	Exp	Exp
	$c^{\mathsf{d}} : \tau$	c	$\uparrow_\tau\ \underline{\text{“}c\text{”}}$	$\Uparrow_\tau\ \ulcorner\langle \underline{\texttt{VAR}} \rangle\urcorner\ (\mathit{lift}_{\texttt{string}}\text{“}c\text{”})$
	$\$_{\mathsf{b}}$	$\lambda x.x$	$\mathit{lift}_{\mathsf{b}}$	$\uparrow_{\bullet\to\bullet}\ \underline{\text{“}\mathit{lift}_{\mathsf{b}}\text{”}}$

Table 1: Instantiation table

```
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 EFullNbE
  infixr 5 -->
  val Ereify_n_exp
  = reify  (((a' --> a') --> (a' --> a')) --> a')
                                                          (* ↓⌜num⌝→• *)
  open RPureNbE
  infixr 5 -->
  val Rreify_n_n
  = reify (((a'-->a') --> (a'-->a')) -->
           ((a'-->a') --> (a'-->a')))                     (* ⇓⌜num→num⌝ *)
in val ge_add
  = Ereify_n_exp (fn m => (Rreify_n_n (fn n =>
    cAdd (m, n))))                                        (* +ch‡ *)
end;
```

The (pretty-printed) result $+_{\mathsf{ch}}{}^{\ddagger}$ is:

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

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 11: Church numerals

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

Example 19. *We revisit the function* `height`*, which appeared in Example 5 on page 11 and Example 7 on page 13. In Figure 12 on the next page 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* $\ulcorner\texttt{height}_{\mathsf{ann}}\urcorner$*. Applying the second Futamura projection as given in Equation (4) on page 23 yields*

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

4.4 Type specification for self-application

The technique developed so far is already sufficient to carry out visualization or the second Futamura projection, at least in an effect-free setting. Still, it requires the user to manually instantiate self-application Equation (3) on page 22 and Equation (4) on page 23, as we have done for all the proceeding examples. In particular, as Example 18 on page 28 demonstrates, one needs to use two different sets of combinators for essentially the same type ($\bullet \to \bullet \to \bullet \to \bullet$ in this case), one for the residualizing instantiation of NbE, and the other for the evaluating instantiation. It would be preferable to package the abstract formulation of Equation (3) on page 22 and Equation (4) on page 23 as program modules themselves, instead of leaving them as templates for the user.

Types are part of the input in both forms of self-application. The user of the module should specify a type τ in a way that is independent of the instantiations; it is the task of the self-application module to choose whether and where to use the residualization instantiation $(\Downarrow^\tau, \Uparrow_\tau)$ or the evaluation instantiation $(\downarrow^\tau, \uparrow_\tau)$. Since different instantiations have different types, the type argument, even in the form of an encoding of the corresponding extraction functions, cannot be abstracted over at the function level. Recall that the type-indexed functions are formed by plugging together combinators. Specifying a type, therefore, amounts to writing down how combinators should be plugged together, leaving the actual definition of the combinators (i.e., an `NBE`-structure) abstract.

To make the above idea more precise, let us consider the example of visualizing the reification functions. The specification of a type τ should consist of not only the type τ itself, but also a functor that maps a `NBE`-structure `NbE` to the appropriate instantiation of the pair $(\langle\underline{\downarrow^\tau}\rangle, \langle\underline{\uparrow_\tau}\rangle)$, which is of type τ `NbE`.rr. This suggests that the type specification should have

```
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                  (* height_ann *)
                  structure D: DYNAMIC
                   sharing type D.SReal = S.SReal) =
 struct
   fun height a z = D.mult (D.lift_real (S.sin a)) z
 end

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

structure GEDynamic: DYNAMIC =                        (* Φ_m on Σ^d *)
  struct
    local open RExp RFullNbE; infixr 5 --> in
      type DReal = Exp
      val mult = reflect (a' --> a' --> a')
                         (VAR (EExp.STR "mult"))
      fun lift_real r = LIT_REAL r
    end
  end
                                                  (* "height_ann" *)
structure ge_height = height_ge(structure S = GEStatic
                                structure D = GEDynamic)
```

Figure 12: Instantiation via functors

```
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                           (* Evaluating instantiation *)
              = P.inp(EFullNbE)
      structure rVIS                         (* Residualizing instantiation *)
              = P.inp(RPureNbE)
      open EFullNbE
      infixr 5 -->
    in
      val vis = reify (eVIS.T_enc --> a')                (* ↓^τ → • (⇓^τ) *)
                      (RPureNbE.reify rVIS.T_enc)
    end
  end
```

Figure 13: Specifying types as functors

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 13).

The example visualization in Figure 10 on page 30 can be now carried out using the type specification given in Figure 14 on the following page.

4.5 Monomorphizing control operators

So far we have shown how to self-apply Pure TDPE. When self-applying Full TDPE, one complication arises: The implementation of Full TDPE uses control operators polymorphically in the definition of reflection, but to determine the residualizing instantiation of a constant, a fixed monomorphic type has to be determined. This section shows how to rewrite the algorithm for full TDPE such that all control operators occur monomorphically.

```
structure a2a : VIS_INPUT =                    (* A type specification *)
  struct
    type 'a vis_type = 'a->'a->'a->'a          (* τ = • → • → • → • *)
    functor inp(NbE: NBE) =                                  (* NbE *)
      struct
        local open NbE  infixr 5 --> in
          val T_enc = a' --> a' --> a' --> a'          (* τ NbE.rr *)
        end
      end
    end

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

Figure 14: Type specification for visualizing $\downarrow^{\bullet\to\bullet}$

Let-insertion via control operators

Full TDPE treats call-by-value languages with computational effects. In this setting, *let-insertion* [3, 19] is a standard partial-evaluation technique to prevent duplicating or discarding computations that have side-effects: All computation that might have effects is bound to a variable and sequenced using the (monadic) let construct. However, when the TDPE algorithm identifies the need to insert a let-construct, it usually is not at a point where a let-construct can be inserted, i.e., a code-generating expression.

Using a technique that originated in continuation-based partial evaluation [24], Danvy [4] solves this problem by using the control operators shift and reset [9]: Intuitively speaking, the operator shift abstracts the current evaluation context up to the closest delimiter reset and passes the abstracted context to its argument, which can then invoke this delimited evaluation context just like a normal function. Formally, the semantics of shift and reset is expressed in terms of the CPS transformation (Figure 15; see Danvy and Filinski [9] and Filinski [14] for more details, and Danvy and Yang [13] for an operational account).

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

where $\langle E\rangle$, "the thunk of E", is shorthand for $\lambda().E$. The use of a thunk here delays the computation of E and avoids the need to implement reset as a macro.

Figure 15: The CPS semantics of shift/reset

With the help of these control operators, Danvy's treatment [4] follows

the following strategy for let-insertion: (1) use reset to 'mark the boundaries' for code generation, i.e., to surround every expression that has type `Exp` and could potentially be a point where let-bindings need to be inserted;[3] (2) when let-insertion is needed, use shift to 'grab the context up to the marked boundary' and bind it to a variable k (thus k is a code-constructing context); (3) apply k to the intended return value to form the body expression of the let-construct, and then wrap it with the let-construct. The new definitions for the reification and reflection functions as given by Danvy are shown in Figure 16; there are two function type constructors: a function type without effects $\tau_1 \to \tau_2$, which does not need let-insertion, and a 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 **let** $\underline{x}$ = t_1 **in** t_2 **end** for `LET (`x`,`t_1`,`t_2`)`; we implement a new TDPE combinator `-!>` in ML for the new type constructor $\xrightarrow{!}$.

$$
\begin{array}{rcll}
\downarrow^{\bullet} e & = & e & \\
\downarrow^{\tau_1 \to \tau_2} f & = & \underline{\lambda\, x}.\mathsf{reset}\langle \downarrow^{\tau_2} (f(\uparrow_{\tau_1} \underline{x}))\rangle & (x \text{ is fresh}) \\
\downarrow^{\tau_1 \xrightarrow{!} \tau_2} f & = & \underline{\lambda\, x}.\mathsf{reset}\langle \downarrow^{\tau_2} (f(\uparrow_{\tau_1} \underline{x}))\rangle & (x \text{ is fresh}) \\
 & & & \\
\uparrow_{\bullet} e & = & e & \\
\uparrow_{\tau_1 \to \tau_2} e & = & \lambda\, x.\uparrow_{\tau_2} (e \mathbin{\underline{@}} (\downarrow^{\tau_1} x)) & \\
\uparrow_{\tau_1 \xrightarrow{!} \tau_2} e & = & \lambda\, x.\mathsf{shift}\,(\lambda\, k.\underline{\mathbf{let}}\ \underline{x'} = e \mathbin{\underline{@}} \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ \mathsf{reset}\langle k(\uparrow_{\tau_2} \underline{x'})\rangle\ \underline{\mathbf{end}}) & \\
 & & & (x' \text{ is fresh})
\end{array}
$$

Figure 16: 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.

[3] An effect-typing system can provide a precise characterization of where reset has to be used. Roughly speaking, an operator reset encloses the escaping control effect introduced by an inner shift. See Filinski's recent work [15] for more details.

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

(x' is fresh)

Now the aforementioned problem is solved, since the return type of shift is fixed to unit—the new definition is a *monomorphized* version of the original.

To show that this change is semantics-preserving, we compare the CPS semantics of the original definition and the new definition.

Proposition 20. *The terms* $[\![\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Here β_v and η_v are respectively the β and η rules in Moggi's computational lambda calculus λ_c [26], i.e., the restricted forms of the usual β rule, $(\lambda x.e')e \sim e'[x := e]$, and of the usual η rule, $\lambda x.ex \sim e$, where the expression e must be a value. These rules are sound for call-by-value languages with computational effects.

Proof. First of all, we abstract out the same computations in the two terms:

$$\begin{array}{rcl} B & \equiv & \lambda\, f.\underline{\mathbf{let}}\ \underline{x'} = e\,\underline{@}\ \downarrow^{\tau_1} x\ \underline{\mathbf{in}}\ f()\ \underline{\mathbf{end}} \\ R & \equiv & \uparrow_{\tau_2} \underline{x'} \\ C[\,] & \equiv & \lambda\, e.\lambda\, x.\mathbf{let}\ x' = \mathsf{new}()\ \mathbf{in}\ [\,]\ \mathbf{end} \end{array}$$

Then

$$\begin{array}{l} \uparrow_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[\mathsf{shift}\,(\lambda\, k.B(\lambda\,().\mathsf{reset}\langle k(R)\rangle))] \\ \uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[(\lambda\,().R)(\mathsf{shift}\,(\lambda\, k.B(\lambda\,().\mathsf{reset}\langle k()\rangle)))] \end{array}$$

Because the CPS transformation is compositional and preserves $\beta_v\eta_v$ equivalence, it suffices to prove that the CPS transformations of the two terms enclosed by $C[\cdot]$ are $\beta_v\eta_v$-equivalent, for all terms B and R. It is a tedious but straightforward check. □

Recently, Sumii [30] pointed out that the reset in the above definition can be removed. The continuation k, being captured by shift, resets the continuation automatically when applied to an argument, which makes the reset in the above redundant, since the argument of k is a value. In contrast, the original definition still requires the reset, since the expression $\uparrow_{\tau_2} \underline{x'}$ might have latent escaping control effect, as in the case where τ_2 is of form $\tau \xrightarrow{!} \tau'$. This simplification improves the performance of TDPE and the generating extension generated by self-application.

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

(x' is fresh)

Proposition 21. *The terms* $[\![\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Proof. We proceed as in the proof of Proposition 20. In particular, using B, R, and $C[\,]$ introduced there, we have that

$$\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2} =_{\beta_v\eta_v} C[(\lambda\,().R)(\mathsf{shift}\,(\lambda\,k.B(\lambda\,().k())))].$$

□

Example 22. *The monomorphized definitions* $\uparrow^{\mathrm{new}}_{\tau_1 \xrightarrow{!} \tau_2}$ *and* $\uparrow^{\mathrm{new}'}_{\tau_1 \xrightarrow{!} \tau_2}$ *of reflection for function types with latent effects are amenable to TDPE itself. Figure 17 on the next page shows the result of visualizing the reification function at the type* $(\bullet \xrightarrow{!} \bullet) \xrightarrow{!} \bullet$*. Note that both* shift *and* reset *have effects themselves; consequently TDPE has inserted let-constructs for the result of visualization. For comparison, we also show the visualization of* $(\bullet \to \bullet) \to \bullet$ *of Pure NbE, which is much more compact.*

The main difference here is the control operators used in Full TDPE, which remain in the result of self-application; later in Section 6, we will see how this difference affects the speedup achieved by the second Futamura projection.

Sum types Full TDPE also treats sum types using control operators; this treatment is also due to Danvy [5]. 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 $\overline{\tau_1}+\overline{\tau_2}$, i.e., a value either of the form **inl** $(v_1 : \overline{\tau_1})$ or **inr** $(v_2 : \overline{\tau_2})$ (for some appropriate v_1 or v_2); on the other hand, both values are needed to have the complete information. Danvy's solution is to "return twice" to the context by capturing the delimited context and applying it separately to **inl** $(\uparrow_{\tau_1}\ e_1)$ and **inr** $(\uparrow_{\tau_2}\ e_2)$; the results are combined using a case-construct which introduces the bindings for e_1 and e_2. Danvy's definition of $\uparrow_{\tau_1+\tau_2}$ is given below:

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

where `Exp` has been extended with constructors for a case distinction and injection functions in the obvious way. Again, the return type of the shift-expression in the above definition is not fixed; an alternative definition is needed to allow self-application.

Following the same analysis as before, we observe that the arguments to k must be one of the two possibilities, **inl** $(\uparrow_{\tau_1}\ e_1)$ and **inr** $(\uparrow_{\tau_2}\ e_2)$, so the information to be passed through the continuation is just the binary

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

$\lambda x_1.$**let** $r_2 = \mathsf{init}()$
$\quad\quad r_3 = \mathsf{new}()$
$\quad\quad r_{11} = \mathsf{reset}\langle$**let** $r_{10} = x_1(\lambda x_5.\mathsf{insertLet}(\underline{r_3}\,\underline{@}\,x_5))$ **in** r_{10} **end**$\rangle$
in $\underline{\lambda\, r_3}.r_{11}$
end

where $\mathsf{insertLet}(E)$ abbreviates the expression

let $x' = \mathsf{new}()$
$\quad\_ = \mathsf{shift}(\lambda k.$**let** $r = k()$ **in** (**let** $\underline{x'} = E$ **in** r **end**)**end**)
in $\underline{x'}$
end

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

$\lambda x_1.$**let** $r_2 = \mathsf{init}()$
$\quad\quad r_3 = \mathsf{new}()$
in $\underline{\lambda\, r_3}.x_1(\lambda x_4.\underline{r_3}\,\underline{@}\,x_4)$
end

Figure 17: Visualizing TDPE with let-insertion

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 **false** as arguments to continuation k (again, Sumii's remark on the redundancy of reset in the program after change applies, and we have dropped the unnecessary occurrences of reset):

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

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

Proposition 23. $[\![\uparrow^{\mathrm{new}}_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *and* $[\![\uparrow_{\tau_1+\tau_2}]\!]_{\mathsf{CPS}}$ *are* $\beta_v\eta_v$*-convertible.*

Using $\uparrow^{\mathrm{new}}_{\tau_1+\tau_2}$ and $\uparrow^{\mathrm{new}'}_{\tau_1 \overset{!}{\to} \tau_2}$ instead of the original definitions provides us with an algorithm for Full TDPE that is amenable to self-application. In the following section, we use self-application of TDPE for compiler generation.

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 this purpose in several instances [4, 5, 11, 12]. Having implemented the second Futamura projection, we can instead *generate* a compiler as the generating extension of an interpreter.

One of the languages for which compilation with TDPE has been studied is *Tiny* [4, 27], 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` that is parameterized over all dynamic constructs. Appendix B.1 gives an overview of Tiny and type-directed partial evaluation of a Tiny interpreter. Compiling Tiny programs by partially evaluating the interpreter `meaning` corresponds to running the trivial generating extension `meaning`†.

Following the development in Section 4.3, we proceed in three steps to generate a Tiny compiler:

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 Table 1 on page 30, thereby creating the GE-instantiation $\overline{\texttt{meaning}}$.

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

Appendix B.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 is a problem for ML-constructs like recursion and case-distinctions over recursive data types. Both have to be rewritten, using fixed-point operators and elimination functions, respectively.

- For every occurrence of a constant in the program, its monotype has to be determined; constants used at more than one monotype give rise to several instances. This is a consequence of performing *type-directed* partial evaluation; for the second Futamura projection, every constant is instantiated with a code-generating function, the form of which depends on the exact type of the constant in question.

program p	static inp. s	specialization time with $p^\dagger$ (s)	specialization time with $p^\ddagger$ (s)	Speedup (ratio)
`meaning`	`factorial`	261.2	194.9	1.34
$\texttt{meaning}_{\mathsf{orig}}$	`factorial`	169.5	99.2	1.71
$+_{\mathsf{ch}}$	80_{ch}	58.45	19.95	2.93

Table 2: Benchmarks: time of specializations (1,000,000 repeated executions)

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 that have to be considered, since functions, products and sums do not require binding-time annotations, and
- TDPE uses the ML type system: Type checking checks the consistency of the binding-time annotations.

6 Benchmarks

6.1 Experiments and results

In Section 3 we claimed that the specialized generating extension $p^\ddagger$ of a program p produced by the second Futamura projection for TDPE is, in general, more efficient than the trivial generating extension $p^\dagger$. In order to assess how much more efficient $p^\ddagger$ is than $p^\dagger$, we performed some benchmarks.

The benchmarks were performed on a 250 MHz Silicon Graphics O_2 workstation using Standard ML of New Jersey version 110.0.3. We display the results in Table 2. In each row of the table, we compare the time it takes to specialize the subject program p with respect to the static input s using two different generating extensions: (1) the trivial generating extension $p^\dagger$ (i.e., directly running TDPE on program p), and (2) the specialized generating extension $p^\ddagger$ (i.e., running the result of the second Futamura projection). We calculate the speedup as the ratio of their running times.

The first row compares the compilers derived from the interpreter `meaning` (see Section 5 and Appendix B); the result shows a speedup of 1.34 for compiling the factorial function. One might wonder, however, whether there is any real gain in using the second Futamura projection: The changes that are necessary to provide the GE-instantiation of `meaning` (replace built-in pattern-matching and recursive function definition of ML with user-defined

fixed-point operators and case operators, respectively—see Section 5) slow down both direct compilation with TDPE and compilation using the specialized generating extension. In fact, as the table's second row shows, direct compilation with the 'original' interpreter $\texttt{meaning}_{\mathsf{orig}}$, i.e., an instantiation of `tiny_pe` rather than `tiny_ge` (cf. Sections 2.2 and 4.3), runs even faster than the specialized generating extension $\texttt{meaning}^{\ddagger}$.

We can do better by eliminating the user-defined fixed point operators and case operators in the result program $\texttt{meaning}^{\ddagger}$, using the built-in constructs.[4] This yields a program that can be understood as the specialized generating extension of the program $\texttt{meaning}_{\mathsf{orig}}$, and we thus call it $\texttt{meaning}_{\mathsf{orig}}{}^{\ddagger}$. The second row of Table 2 on the preceding page shows that running $\texttt{meaning}_{\mathsf{orig}}{}^{\ddagger}$ gives a speedup of 1.71 over running the original program $\texttt{meaning}_{\mathsf{orig}}$. The speedup over the direct compilation using the original interpreter here is, in practice, more relevant than the speedup of the benchmark shown in the first row.

The benchmark in the third row compares the generating extensions of an effect-free function, the addition function $+_{\mathsf{ch}}$ for Church numerals. Because the function is free of computational effects (we assume that its argument function is also effect-free), we can specialize Pure TDPE instead of Full TDPE in the second Futamura projection. The speedup of running the specialized generating extension over direct partial evaluation is consistently around 3 (shown with Church numeral 80_{ch}).

6.2 Analysis of the result

Overall, the speedup of the second Futamura projection with TDPE is disappointing compared to the typical order-of-magnitude speedup achievable in traditional partial evaluation [22]. This, on the other hand, reflects the high efficiency of TDPE, which carries out static computations by evaluation rather than symbolic manipulation. Turning symbolic manipulation (i.e., interpretation) into evaluation is one of the main goals one hopes to achieve by specializing a syntax-directed partial evaluator. Since TDPE does not have much interpretive overhead in the first place, the further speedup is bound to be lower.

Logically, the next question to ask—for a better understanding of how and when the second Futamura projection could effectively speedup the TDPE process—is what cost of TDPE can or cannot be removed by using the self-application. The higher-order nature of the TDPE algorithm blurs the boundaries between the various components that contribute to the running

[4] Removing the user-defined fixed point operator and case operators can be carried out automatically by (1) incorporating TDPE with patterns as generated bindings, and (2) systematically changing the residualizing instantiations for the fixed point and case operators used. Danvy and Rhiger [11] achieved a similar effect in TDPE for Scheme, using Scheme macros.

time of the specialization; we can only roughly divide the cost involved in performing TDPE as follows:

1. Cost due to computation in the extraction function $\downarrow^\tau$, namely function invocations of reification and reflection for subtypes of τ, name and code generation and, in the case of Full TDPE, the use of control operators shift and reset.

2. Cost due to computation in the residualizing instantiation $\overline{ps}$ of input program and static input, namely, apart from static computation, the invocation of reflection by code-generating versions of dynamic constants (see Section 2.1).

3. Cost due to reducing extra redexes formed by the interaction of $\downarrow^\tau$ and $\overline{ps}$ in $\downarrow^\tau \overline{ps}$.

Of the cost due to computation in the extraction function, only the one caused by function invocations can be eliminated: All other computations have to be performed at specialization time. This optimization amounts to function inlining. Similarly, for the cost associated with the residualizing instantiation, inlining can be performed for the code-generating versions of dynamic constants and their calls to the reflection function. Finally, the extra redexes formed by the interaction of the extraction function and the residualizing instantiation can be partly reduced by the specialization.

In Full TDPE, the somewhat time-consuming control operators dominate the cost of extraction algorithm; the low speedup of specializing Full TDPE (the first two benchmarks) as opposed to that of specializing Pure TDPE (the third benchmark), we think, are mainly due to the fact that these control operators cannot be eliminated. Furthermore, in the case of the Church addition function, the program is a higher-order pure λ-term, which usually "mixes well" with the extraction function, in the sense that many extra redexes are formed by their interaction.

Do certain implementation-related factors, such as the global optimizations of the ML compiler we used and the fact that we are working in a typed setting, give positive contribution to the speedup? In our opinion, the help is minimal, if not negative. First, the specialization carried out by the self-application, where the program is given a trivial BTA (Section 3.1), has an effect similar to a good global inliner. Therefore, the global optimization of the ML compiler, especially the inlining optimization, should only reduce the potential speedup of the specialization. Second, working in a typed setting does complicates the type specification and the parameterization (Section 4.4), but it does not incur extra cost at runtime when using TDPE—the instantiation through ML functors happens at compile time; furthermore, the need to parameterize over built-in constructs such as fixed point operators and pattern matching is present also in an untyped setting.

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.

To hand-write a cogen and to formally prove its correctness at the same time, one possibility is to start with a partial evaluator and rewrite it into the desired generating extension in several steps, such as the use of higher-order abstract syntax and deforestation in Thiemann's work [31]. Correctness follows from showing the correctness of the partial evaluator and the correctness of each of these steps. In contrast, for generating extensions produced with the second Futamura projection, the implementation is produced automatically, and correctness follows immediately from the correctness of the partial evaluator. Often, however, this conceptual simplicity is compromised by (1) the complications in using self-application, and (2) the need to make the partial evaluator self-applicable and prove the necessary changes to be meaning preserving. In the case of TDPE, the implementational effort for writing the GE-instantiation of the object program is similar in level to that of the hand-written cogen approach, but the only change to the TDPE algorithm itself is the transformation described and proven correct in Section 4.5.

The third Futamura projection states that specializing a partial evaluator with respect to itself yields an efficient generating-extension generator. The type-indexed nature of TDPE makes it challenging, 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 would be obtained.

At the current stage, our contribution seems to be more significant at a conceptual level, since the speedup achieved by using the generated generating extensions is rather modest. However we observed that a higher speedup can be achieved for more complicated type structures, especially in a setting with no or few uses of computational effects; this suggests that our approach to the second Futamura projection using TDPE might find more practical applications in, e.g., the field of type theory and theorem proving.

The technical inconveniences mentioned in Section 5 are clearly an obstacle for using the second Futamura projection for TDPE (and, to a lesser extent, for using TDPE itself). A possible solution is to implement a translator from the two-level language into ML, thus handling the mentioned technicalities automatically. Of course, such an approach would sacrifice the flexibility of TDPE of allowing the use of all language constructs in the static part of the subject program. Even so, TDPE would still retain a distinct flavor when compared to traditional partial evaluation techniques:

Only those constructs not handled automatically by TDPE, i.e., constants, need to be binding-time annotated; other constructs, such as function application and function abstraction, always follow their standard typing rules from typed lambda calculi. This simplifies the binding-time analysis considerably and often makes binding-time improvements, e.g, eta-expansion, unnecessary, which was one of the original motivations of TDPE [5, 10].

Acknowledgments

At an early stage both Olivier Danvy and Morten Rhiger [29] 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 4.5).

We are grateful to Daniel Damian, Olivier Danvy, Andrzej Filinski, Lasse R. Nielsen, Morten Rhiger, our anonymous referees from HOSC and PEPM'00, and our editor Julia Lawall for their numerous constructive 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] Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel Hermenegildo and Jaan Penjam, editors, *Sixth International Symposium on Programming Language Implementation and Logic Programming*, number 844 in Lecture Notes in Computer Science, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.

[3] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[4] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation, Proceedings*, number 1110 in Lecture Notes in Computer Science, pages 73–94. Springer-Verlag, 1996.

[5] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on*

Principles of Programming Languages, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[6] 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, Aalborg, Denmark, 1998. Springer-Verlag.

[7] Olivier Danvy. Type-directed partial evaluation. In *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. xtended version available as BRICS technical report LN-98-3.

[8] Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Gothenburg, Sweden, May 8–9, 1998), number NS-98-8 in Note Series, Department of Computer Science, University of Aarhus, May 1998. BRICS.

[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, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

[11] 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.

[12] 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.

[13] Olivier Danvy and Zhe Yang. An operational investigation of the CPS hierarchy. In S. Doaitse Swierstra, editor, *Proceedings of the Eighth European Symposium on Programming*, number 1576 in Lecture Notes in Computer Science, pages 224–242, Amsterdam, The Netherlands, March 1999. Springer-Verlag.

[14] 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.

[15] Andrzej Filinski. Representing layered monads. In Alex Aiken, editor, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, January 1999. ACM Press.

[16] 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.

[17] 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.

[18] Bernd Grobauer and Zhe Yang. Source code for the second Futamura projection for type-directed partial evaluation in ML, 2000. Available from `http://www.brics.dk/~tdpe/second_FP/sources.tgz`.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[24] Julia L. Lawall and Olivier Danvy. Continuation-based partial evaluation. In Carolyn L. Talcott, editor, *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. VII, No. 3, Orlando, Florida, June 1994. ACM Press.

[25] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[26] Eugenio Moggi. Computational lambda-calculus and monads. In Rohit Parikh, editor, *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[27] 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.

[28] 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*, BRICS technical report BRICS-NS-99-1, pages 25–29, Department of Computer Science, University of Aarhus, 1999. BRICS.

[29] 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.

[30] Eijiro Sumii, 2000. Email exchange, February 2000.

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

[32] 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 Notation and symbols

Font Conventions

p, s, d, ...	terms (one-level or two-level)	3
x, y, z, ...	variable names	3
$\underline{x}$, $\underline{@}$, $\underline{\textbf{let}}$,...	constructors for code representation (of type `Exp`)	8

Language

Σ	signature	16
$\mathcal{I}$	interpretation	16
$(\Sigma, \mathcal{I})$	language specification	16
$t : \sigma$ or $\Gamma \vdash_\Sigma t : \sigma$	typing judgment	16
$[\![\cdot]\!]$, $[\![\cdot]\!]^{\mathcal{I}}$	meaning function	3,16
$t[x := t']$	substitution of t' for x in t	
Φ	instantiation	17
$t\{\Phi\}$	application of Φ to t	17

Two-level language

$\Sigma^2 = \Sigma^{\mathfrak{s}}, \Sigma^{\mathfrak{d}}$	two-level signature with static part $\Sigma^{\mathfrak{s}}$ and dynamic part $\Sigma^{\mathfrak{d}}$	17
$c^{\mathfrak{s}}$, $\mathsf{b}^{\mathfrak{s}}$, ...	static constants and base-types (part of $\Sigma^{\mathfrak{s}}$)	17
$c^{\mathfrak{d}}$, $\mathsf{b}^{\mathfrak{d}}$, ...	dynamic constants and base-types (part of $\Sigma^{\mathfrak{d}}$)	17
$\$_{\mathsf{b}}$	lifting function on base-type b (part of $\Sigma^{\mathfrak{d}}$)	17
$\mathcal{I}^{\mathfrak{s}}$	interpretation of static signature $\Sigma^{\mathfrak{s}}$	17
$\mathcal{I}^{\mathfrak{d}}$	interpretation of dynamic signature $\Sigma^{\mathfrak{d}}$	17
$PL^2 = (\Sigma^2, \mathcal{I}^{\mathfrak{s}})$	two-level language (fixing only the interpretation of $\Sigma^{\mathfrak{s}}$)	17
$PL = (\Sigma^{PL}, \mathcal{I}^{PL})$	one-level language associated with PL^2	17
$NF(e)$	static normal-form of two-level term e	18
t_{ann}	binding-time annotated term (a two-level term)	20
$\underline{\langle \cdot \rangle}$	trivial binding-time annotation (defined by $\Phi_{\underline{\langle\rangle}}$)	22,24
$\tilde{\cdot}$	evaluating instantiation (defined by $\Phi_\sim$)	18
$\overline{\cdot}$	residualizing instantiation (defined by $\Phi_{\overline{\ }}$)	19
$\overline{\underline{\cdot}}$	GE-instantiation (defined by $\Phi_{\overline{\underline{\ }}}$)	26

PE-specific notation

PE	code of a partial evaluator	3
p_s	result of specializing program p to input s	3
$p^\dagger$	trivial generating extension of program p	23
$p^\ddagger$	efficient generating extension of program p	23

TDPE-specific notation

$\downarrow^\tau$	reification function at type τ	9
$\uparrow_\tau$	reflection function at type τ	9
$\bullet$	abbreviation for any base type	9
$\Downarrow^\tau$	abbreviation for $\overline{\langle \downarrow^\tau \rangle}$	22

$\Uparrow_\tau$	abbreviation for $\overline{\langle \uparrow_\tau \rangle}$	22

ML function symbols

init	initializes the name generator	28
new	generates a new name	28
shift, reset	control operators	35

B Compiler generation for Tiny

B.1 A binding-time-separated interpreter for Tiny

Paulson's Tiny language [27] is a prototypical imperative language—the BNF of its syntax is given in Figure 18. Figure 19 on the following page 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

Experiments in type-directed partial evaluation of a Tiny interpreter with respect to a Tiny program [4, 5] used an ML implementation of a Tiny interpreter (Figure 20 on page 54): For every syntactic category a meaning function is defined—see Figure 21 on page 55 for the ML data type 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

```
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

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 on page 9 and made precise in Definition 10 on page 19, 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
```

B.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 Table 1 on page 30, thereby creating the GE-instantiation "`meaning`"

3. use the GE-instantiation "`meaning`" to perform the second Futamura projection

The following two sections describe the first two steps in more detail. Once we have a GE-instantiation, the third step is easily carried out with the help of an interface similar to the one for visualization described in Section 4.4.

B.3 "Full parameterization"

Following Section 4.3 we re-implement the interpreter inside a functor to parameterize over *both* static and dynamic base types and constants. Note, however, that the original implementation of Figure 20 on page 54 makes use of recursive definitions and case distinctions; both constructs cannot be parameterized over directly. Hence we have to express recursive definitions with a fixed point operator and case distinctions with appropriate elimination functions. Consider for example case distinction over `Expression`; Figure 22 on page 55 shows the type of the corresponding elimination function.

The resulting implementation is sketched in Figure 23 on page 56. 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 on page 57. 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 ($\texttt{int}^{\texttt{S}}$) and dynamically ($\texttt{int}^{\texttt{D}}$)—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 on page 57 shows a portion of signature `STATIC`: The polymorphic type of `caseExpression` (Figure 22 on page 55) 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.[5] For base types which

[5]Note that static base types appear also in the signature of dynamic constructs, because

occur both in signatures `STATIC` and `DYNAMIC`, sharing constraints have to be declared in the interface of functor `tiny_ge` (Figure 24 on page 57).

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 into a type specification by hand. This transcription is straightforward, because the type specifications of TDPE and the output of ML type inference are very much alike.

B.4 The GE-instantiation

After parameterizing the interpreter as described above, we are in a position to either run the interpreter by using its evaluating instantiation (see Definition 9 on page 18), perform type-directed partial evaluation by employing the residualizing instantiation (Definition 10 on page 19), or carry out the second Futamura projection with the GE-instantiation (Definition 17 on page 26). Section 4.3 shows how the static and dynamic constructs have to be instantiated in each case. For the GE-instantiation, all base types become `Exp`; static and dynamic constants are instantiated with code-generating functions. The latter are constructed using the evaluating and the residualizing instantiation of reflection, respectively. Because the signatures `STATIC` and `DYNAMIC` hold the precise type at which each constant is used, it is purely mechanical to write down the structures needed for the GE-instantiation.

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.

```
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                          (* Type abbreviation *)
  = Expression -> ((Int_s -> 'a) *
                   (Identifier -> 'a) *
                   (Bop * Expression * Expression -> 'a)
                   ) -> 'a

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

⋮
                    (* Declaration of elimination function for expressions *)
val case_Expression: case_Exp_res_type case_Exp_type

⋮
```

Figure 25: Excerpts from signature `STATIC`

Recent BRICS Report Series Publications

RS-00-44 **Bernd Grobauer and Zhe Yang.** ***The Second Futamura Projection for Type-Directed Partial Evaluation*****. December 2000. To appear in** ***Higher-Order and Symbolic Computation*****. This revised and extended report supersedes the earlier BRICS report RS-99-40 which in turn was an extended version of Lawall, editor,** ***ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation,*** **PEPM '00 Proceedings, 2000, pages 22–32.**

RS-00-43 **Claus Brabrand, Anders Møller, Mikkel Christensen, Ricky, and Michael I. Schwartzbach.** ***PowerForms: Declarative Client-Side Form Field Validation*****. December 2000. 21 pp. To appear in** ***World Wide Web Journal*****, 4(3), 2000.**

RS-00-42 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***The <bigwig> Project.*** **December 2000. 25 pp.**

RS-00-41 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***The DSD Schema Language and its Applications*****. December 2000. 32 pp. Shorter version appears in Heimdahl, editor,** ***3rd ACM SIGSOFT Workshop on on Formal Methods in Software Practice*****, FMSP '00 Proceedings, 2000, pages 101–111.**

RS-00-40 **Nils Klarlund, Anders Møller, and Michael I. Schwartzbach.** ***MONA Implementation Secrets*****. December 2000. 19 pp. Shorter version appears in Daley, Eramian and Yu, editors,** ***Fifth International Conference on Implementation and Application of Automata*****, CIAA '00 Pre-Proceedings, 2000, pages 93–102.**

RS-00-39 **Anders Møller and Michael I. Schwartzbach.** ***The Pointer Assertion Logic Engine*****. December 2000. 23 pp. To appear in** ***ACM SIGPLAN Conference on Programming Language Design and Implementation,*** **PLDI '01 Proceedings, 2001.**

RS-00-38 **Bertrand Jeannet.** ***Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Synchronous Programs*****. December 2000. 44 pp.**