Two Flavors of Offline Partial Evaluation

Simon Helsen[1] and Peter Thiemann[2]

[1] Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Sand 13, D-72076 Tübingen, Germany
helsen@informatik.uni-tuebingen.de
[2] University of Nottingham
Department of Computer Science
University Park
Nottingham NG7 2RD, England
pjt@cs.nott.ac.uk

Abstract. Type-directed partial evaluation is a new approach to program specialization for functional programming languages. Its merits with respect to the traditional offline partial evaluation approach have not yet been fully explored. We present a comparison of type-directed partial evaluation with standard offline partial evaluation in both a qualitative and quantitative way. For the latter we use implementations of both approaches in Scheme. Both approaches yield equivalent results in comparable time.

1 Introduction

Partial evaluation is a technique for automatically specializing programs. One approach is *offline* partial evaluation where specialization is entirely driven by the results of a program analysis. We consider two offline frameworks which are superficially quite different.

In the *traditional approach* the program analysis is a *binding-time analysis*, an abstraction of the semantics of specialization. A binding-time analysis determines for each program point whether the specializer should generate code or execute it.

An offline partial evaluator is then either an interpreter of annotated programs or it follows the approach of *writing cogen by hand* [34]. Self-application of the partial evaluator improves the performance of the interpreter-based alternative. It can transform the annotated program into a dedicated program generator. The *cogen-by-hand* alternative performs this step directly, thus avoiding any interpretive overheads. The latter alternative is the current state of the art in partial evaluation [1, 5, 13, 22, 43].

In the *type-directed approach* [15] it is only the type of the program (and its free variables) that drives the specialization. It avoids interpretation completely by applying to compiled programs without necessarily requiring access to their source, as long as their types are known. It guarantees that the specialized program is in normal form.

The aim of this work is to clarify the relationship between these two approaches. Some of the issues considered have already been touched upon in Danvy's article "Type-directed Partial Evaluation" [15], but some delicate issues were left implicit. Therefore, we have set out to clarify the principal differences and commonalities of the two approaches and we have performed some practical experiments using representative implementations of both approaches.

For the traditional partial evaluators, we have used the PGG system [47]. It is an implementation of the *cogen-by-hand* approach for the Scheme language.

For type-directed partial evaluation (TDPE), we have used Danvy's implementation for Scheme [14, 15] and our own reimplementation in Scheme to perform experiments with self-application of TDPE.

Although both systems are based on Scheme, the two approaches are applicable to other functional languages like ML and Haskell.

Our experiments concentrate on compilation, the flagship application of partial evaluation. Compiling with a partial evaluator means to specialize an interpreter for a programming language L with respect to an L-program. We call such an interpreter as well as any program submitted to a specializer the *subject program*. We have found that:

- both approaches to specialization require changes to the subject programs to specialize well;
- if the subject program has a simple type and polyvariant program point specialization is not required then TDPE can simulate traditional partial evaluation without change, otherwise if the subject program is untyped some transformations are required to hide the untyped parts from TDPE;
- in all cases, a traditional partial evaluator can achieve the result of a specialization with TDPE, subject to improving binding times using eta-expansion [19] and repeating the binding-time analysis;
- both specialization methods have comparable runtimes (excluding type inference, compilation, etc).

These observations hold for TDPE as originally defined [15]. Recently, Danvy and others [16, 42] have defined online variants of TDPE that are strictly more powerful and yield strictly better results than traditional methods. To match these results, a traditional partial evaluator must revert to multiple passes including aggressive post-processing [41].

The rest of the paper is structured as follows. In Sec. 2 we introduce traditional partial evaluation as well as TDPE. Section 3 formulates important aspects of specialization and puts them into perspective with respect to both approaches. Section 4 shows how both approaches relate to each other in terms of an eta-expanding binding-time analysis. Section 5 analyzes the run-time behavior of both approaches.

Throughout, we assume familiarity with basic notions of partial evaluation [12, 31].

$$\begin{array}{lll} \text{expressions} & \mathit{Expr} \ni & e ::= x \mid c \mid e@e \mid \lambda x.e \\ \text{types} & \mathit{Type} \ni & t ::= b \mid t \rightarrow t \\ \text{reduction} & & (\lambda x.e_1)@e_2 \;\mapsto\; e_1\{x := e_2\} \end{array}$$

Fig. 1. The simply-typed lambda calculus

$$\begin{array}{lll} \text{two-level expressions} & \mathit{TLT} \ni & e ::= x \mid e \,\overline{@}\, e \mid \overline{\lambda}\, x.e \mid e \,\underline{@}\, e \mid \underline{\lambda}\, x.e \\ \text{two-level types} & & t ::= \overline{b} \mid t \overline{\rightarrow} t \mid \underline{b} \mid t \underline{\rightarrow} t \\ \text{static reduction} & & (\overline{\lambda}\, x.e_1) \,\overline{@}\, e_2 \;\mapsto\; e_1\{x := e_2\} \end{array}$$

$$\vdash_{dyn} \underline{b} \qquad \vdash_{dyn} t_2 \underline{\rightarrow} t_1$$

$$\vdash_{wft} \overline{b} \qquad \vdash_{wft} \underline{b}$$

$$\frac{\vdash_{wft} t_2 \quad \vdash_{wft} t_1}{\vdash_{wft} t_2 \overline{\rightarrow} t_1} \qquad \frac{\vdash_{wft} t_2 \quad \vdash_{wft} t_1 \quad \vdash_{dyn} t_2 \quad \vdash_{dyn} t_1}{\vdash_{wft} t_2 \underline{\rightarrow} t_1}$$

$$A\{x : t\} \vdash_{bta} x : t$$

$$\frac{A\{x : t_2\} \vdash_{bta} e : t_1 \quad \vdash_{wft} t_2 \overline{\rightarrow} t_1}{A \vdash_{bta} \overline{\lambda}\, x.e : t_2 \overline{\rightarrow} t_1} \qquad \frac{A \vdash_{bta} e_1 : t_2 \overline{\rightarrow} t_1 \quad A \vdash_{bta} e_2 : t_2}{A \vdash_{bta} e_1 \,\overline{@}\, e_2 : t_1}$$

$$\frac{A\{x : t_2\} \vdash_{bta} e : t_1 \quad \vdash_{wft} t_2 \underline{\rightarrow} t_1}{A \vdash_{bta} \underline{\lambda}\, x.e : t_2 \underline{\rightarrow} t_1} \qquad \frac{A \vdash_{bta} e_1 : t_2 \underline{\rightarrow} t_1 \quad A \vdash_{bta} e_2 : t_2}{A \vdash_{bta} e_1 \,\underline{@}\, e_2 : t_1}$$

Fig. 2. The simply-typed two-level lambda calculus

2 Two paradigms for specialization

This section presents the kernels of the two paradigms for specialization. To simplify our later comparison, the formal presentation focuses on a simply-typed lambda calculus with full β reduction (see Fig. 1). We omit the typing rules since they are standard [37].

To discuss specialization, we employ the two-level call-by-name lambda calculus [24,38] defined in Fig. 2. Overlining denotes *static* specialization-time constructs and underlining denotes *dynamic* run-time constructs.

To discuss implementations of specialization, we augment the simply-typed lambda calculus with a datatype *Expr* for generated or specialized expressions. The operations on *Expr* are *mkLam* : *Var* $\times$ *Expr* $\rightarrow$ *Expr*, which constructs a lambda expression, and *mkApp* : *Expr* $\times$ *Expr* $\rightarrow$ *Expr*. In addition, the impure operator *gensym* : *Unit* $\rightarrow$ *Var* generates a new variable every time it is called. Most of the time we are using the derived operations shown in Fig. 3.

The specifications are for specializing a call-by-name lambda calculus. Specializing the call-by-value lambda calculus or the computational lambda calculus (call-by-value with certain side effects) requires identical changes to both specifications.

$$\hat{\lambda}\, f \quad \equiv let\ y = gensym\ ()\ in\ mkLam(y, f y)$$
$$f \,\hat{@}\, a \equiv mkApp(f, a)$$

Fig. 3. Derived operations on *Expr*

$$cogen : TLT \rightarrow Impl$$

$$cogen(x) \equiv x$$

$$cogen(\overline{\lambda}\, x.e) \equiv \lambda\, x.cogen(e) \qquad cogen(\underline{\lambda}\, x.e) \equiv \hat{\lambda}\, \lambda\, x.cogen(e)$$
$$cogen(f \,\overline{@}\, a) \equiv (cogen(f))(cogen(a)) \qquad cogen(f \,\underline{@}\, a) \equiv (cogen(f)) \,\hat{@}\, (cogen(a))$$

Fig. 4. Cogen-based offline partial evaluation [43]

2.1 Cogen-based offline partial evaluation

A natural approach to implementing a *cogen-by-hand* partial evaluation system is to provide an implementation of each of the constructs of the two-level lambda calculus [22, 43]. In this case, the generation of the dedicated specializer boils down to a trivial translation that replaces each construct in a term of the two-level calculus by its implementation.

Figure 4 shows such an implementation in terms of a mapping *cogen* from two-level expressions *TLT* to an implementation language *Impl*. The $\hat{\lambda}$ combinator interprets the dynamic lambda using a higher-order syntax representation, the $\hat{@}$ combinator interprets dynamic application. The static constructs map into the corresponding constructs of the implementation language.

2.2 Type-directed partial evaluation

The heart of an implementation of TDPE is a pair of mutually recursive functions, *reify* and *reflect*, shown in Fig. 5. *Reify*, written as $\downarrow^t v$, accepts a type t and a value v of type t and converts it into an expression of type t. *Reflect*, written $\uparrow_t e$, accepts a type t and an expression e of type t and converts it into a value of type t. *Type* denotes simple types, *Value* stands for (compiled) values, *Expr* denotes (specialized) expressions and *TLT* are two-level expressions.

Reify at a function type constructs a lambda expression and applies the corresponding function v from the compiled program to the freshly generated variable. *Reflect* constructs an application expression and wraps it into a function to transport it to the correct place in the compiled program.

$$\downarrow : Type \to Value \to Impl \qquad \uparrow : Type \to Expr \to Impl$$

$$\begin{array}{lll} \mathit{reify} & \downarrow^{b} v & = v \\ & \downarrow^{t_1 \to t_2} v & = \mathit{let}\ y = \mathit{gensym}\ ()\ \mathit{in}\ \mathit{mkLam}(y, \downarrow^{t_2} (v(\uparrow_{t_1} y))) \\ \mathit{reflect} & \uparrow_{b} e & = e \\ & \uparrow_{t_1 \to t_2} e & = \lambda\, v.\ \uparrow_{t_2} (\mathit{mkApp}(e, \downarrow^{t_1} v)) \end{array}$$

Fig. 5. Specification of TDPE [15]

3 Aspects of specialization techniques

Each of the following subsections considers one aspect of offline specialization techniques and comments on its significance for the traditional as well as for the type-directed approach.

3.1 Types

Types play an important role for both approaches [15, p.247, sec.2]. TDPE specializes by normalizing a compiled object with respect to its simple type. In practice, the specialized code has a simple type while the computation at specialization time can be untyped. The source program has to be closed with respect to primitive operations and the dynamic values because that is the only way their types can be specified. In contrast to many traditional systems, TDPE does not admit recursive types since there is no general way to reflect dynamic values.

In many traditional systems, the binding-time analysis imposes a certain type discipline upon the language, mostly based on simple typing with partial types and recursion (Lambdamix [24], Henglein's analysis [27], Similix 5.0 [9], and in particular the PGG system [47]). The binding-time analysis defers untypable expressions to run-time. Furthermore, all expressions that depend on dynamic data are also made dynamic. In consequence, the specialization-time computation has a simple type with recursion whereas the specialized program is unconstrained: it can be arbitrarily untyped and/or impure.

In principle, a binding-time analysis could also be based on more powerful type systems. For example, systems with subtyping [45], soft typing [11], or dynamic typing [28] could provide a suitable basis.

While the type of the subject program is usually fixed in the traditional approach, this need not be the case for TDPE. This flexibility is due to the interpretive nature of the *reify*/*reflect* pair of functions, both of which interpret their type argument. In terms of the compilation paradigm, it means that *the type of the interpreter may depend on the type of its source program.* This has two consequences:

1. it is possible to perform "simple type specialization" [17] which can remove type tags in a typed interpreter during specialization;

2. it is no longer sufficient to know the type of the subject program (interpreter) before starting a specialization, but it is necessary to know the *type of the subject program applied to the static input*, that is, the type of the interpreter applied to its source program. Effectively, it boils down to perform type inference of the source program before being able to compile it. In general, this type inference cannot be done by the compiler which compiles the interpreter, but there must be a separate program to type check the source program.

3.2 Polyvariance and program point specialization

Polyvariant specialization means:

> Any program point in the source program gives rise to zero or more program points in the specialized program [10, 23].

Both techniques, traditional partial evaluators and TDPE, perform polyvariant specialization in this sense.

A related, but different issue is concerned with *program point specialization*. This technique uses memoization points to avoid code duplication and potentially infinite specialization. The specialized programs are sets of mutually recursive function definitions.

Program point specialization is a standard feature of traditional partial evaluators [6, 8], which has not yet been achieved for TDPE. It gives raise to a number of problems, for example, traditional systems must use a closure-based representation for functions to be able to compare them.

3.3 Continuation-based reduction

Continuation-based reduction is a commonly used name for a contextual binding-time-improving transformation which takes place during specialization [7]. Continuations are one implementation technique for it. Its main use is the implementation of non-standard reductions in two-level calculi and of dynamic choice of static values [20]. Both systems make use of the same technique to guarantee sound specialization in the presence of computational effects [26, 35]. Both systems have options to switch this feature off for selected operators. For these operators, the systems perform a call-by-name transformation (i.e., the operator has no side effects).

3.4 Computational effects

There are two dimensions of computational effects to consider: effects at specialization time and effects in the specialized program.

Since TDPE does not constrain the computation at specialization time, side effects during specialization are permitted as long as they respect the typing properties. Recent results enable traditional systems to perform impure computations at specialization time, too [44]. Two important problems here are side effects under dynamic control and polyvariant program point specialization, which are discussed elsewhere in this paper.

Both approaches handle computational effects in the specialized program in essentially the same way [26,35]. The necessary modifications to Figures 4 and 5 are almost identical [14,46].

3.5 Compilation vs. interpretation

TDPE as described by Danvy [15] applies to a compiled program and hence performs specialization at full compiled speed. Indeed, this observation is one of Danvy's key contributions because the idea of normalization by evaluation has been developed in an interpretive setting [4].

A traditional partial evaluator accesses the source program to perform the binding-time analysis. Then it interprets the resulting annotated program. A cogen-based partial evaluator turns the annotated program into a program generator as indicated by the *cogen* function. The program generator has (assuming a monovariant binding-time analysis) about the same size as the original program and the static parts run at full compiled speed, since their translation is trivial.

Specialization with either system involves compilation:

- The source program must be compiled to be amenable to TDPE.
- In a cogen-based system, the *cogen* function maps a binding-time annotated program to a program generator. The generator is then compiled before it can perform specialization.

There is a difference in the kind of program that is compiled in the two systems. In the cogen-based system, the compiled program consists of the static part of the subject program interspersed with syntax constructors, which generate code during specialization. TDPE, on the other hand, uses the original program in compiled form. This may result in a more optimized compiled program.

3.6 Termination

Neither TDPE nor traditional systems can guarantee the termination of specialization if there are static fixpoints present in the subject program. However, if the subject program is restricted to the simply-typed lambda calculus (without recursion) then TDPE is sure to terminate, by the strong normalization property of the calculus. A traditional system can employ an extra program analysis which guarantees the termination of specialization [2,29].

3.7 Binding-Time Improvements

A *binding-time improvement* is a program transformation geared at improving the effectiveness of specialization. This goal is usually achieved by using special programming idioms to separate the binding times.

Traditional specializers require binding-time improvements [31] whereas TDPE does not [15, 41]. However, achieving satisfactory results is not trivial using TDPE, either. It is still necessary to address some issues.

Staging discipline The subject program must obey a *staging discipline*. It means that the subject program must compute all values that we expect to see in the specialized programs. This is non-obvious so we illustrate it using an example.

Returning to the compilation application, consider the specialization of an environment lookup $\rho(v)$ in an interpreter. Obviously, the name v of the variable should disappear in the specialized (compiled) program. Instead, we expect something like `lookup store 5` as compiled code. But to this end the interpreter must actually compute the index/address `5` and perform the environment lookup in two stages: in the first (static) stage, compute the index using an additional (static) environment; in the second (dynamic) stage, perform `lookup store index`. This is quite similar to separating binding times, which is what binding-time improvements are about.

The work of Harrison and Kamin [25] provides some more evidence for this fact. For example, they must precompute some labels to compile conditional jumps. Their solution is to have the interpreter interpret the code in two passes: in the first pass, it computes the addresses, and then it "snaps back" to perform the actual computation in the second pass.

Dynamic recursion A program contains dynamic recursion if there is at least one loop that is controlled by run-time (i.e., dynamic) data. This is a frequent phenomenon, since most non-trivial programs contain dynamic recursion. For instance, an interpreter will interpret loops or recursive functions in its source language by dynamic recursion. Traditional partial evaluation memoizes calls to the specializer at points that may give rise to dynamic recursion: dynamic conditionals and dynamic lambda abstractions [8]. Thus it creates a set of top-level mutually recursive functions, one for each memoized call to the specializer.

This approach is currently not possible with TDPE. Instead, the general idea to deal with dynamic recursion [15] is to provide a fixed-point combinator at a suitable type and abstract the subject program over the fixed-point operator. This may not be easy as demonstrated with the following example, again dealing with compilation.

Suppose we want to compile the language Mixwell [32] which provides mutually recursive of first-order functions. The typical semantics of such a language includes semantic equations $\mathcal{E} : \mathit{Exp} \to \mathit{FunctionEnv} \to \mathit{ValueEnv} \to \mathit{Value}$ for expressions and goes on to provide the meaning of a program $f_i(\overline{x_i}) = e_i$ as the environment

$$\mathit{fix}\ \lambda\,\psi \in \mathit{FunctionEnv}.[f_i \mapsto \lambda\,\overline{y_i}.\mathcal{E}[\![e_i]\!]\psi\emptyset[\overline{x_i \mapsto y_i}]]$$

It turns out that it is not straightforward to come up with a suitable fixed-point combinator *fix*.

One strategy (suggested by Danvy[1]) is to implement the function environment ψ as a function and use the fixed-point combinator at a different type for each program: this turns out to be problematic because the type of ψ must reflect the arity of each f_i. In other words, the type of the result of ψf_i depends on the particular argument f_i. Unfortunately, this kind of type is currently outside the scope of TDPE, unless we employ the trick to hide the actual arity of the functions inside of some kind of "dynamic" type. To do so, we have to rewrite the interpreter by including coercions into and out of the dynamic type. But still, the type of ψ—and hence of *fix*—depends on the number of functions f_i in the source program.

If we take this last idea to the extreme, we implement the function environment ψ as an abstract type with lookup and update functions. This works satisfactory for the Mixwell interpreter, but it is still necessary to insert coercions. In this case, the most natural implementation of ψ is a list and the implementation of the fixed-point combinator uses an impure update operation. Anyway, we must split the function environment in the interpreter into a compile-time and a run-time part to achieve a static lookup, just like explained above for the variable lookup.

Dynamic conditional On a smaller scale, a similar problem exists for dynamic conditionals. In the traditional approach, the specializer inserts a memoization point at a dynamic conditional. This is not possible with TDPE.

In principle, there is no problem with handling dynamic conditionals in TDPE [15]. However, the solution can give raise to code duplication which is why most examples use a workaround. The trick is to hide the conditional inside a primitive function and abstract it. Clearly, to avoid changing the semantics in a call-by-value language, such a primitive must work at a functional type.

4 Qualitative comparison

In this section, we investigate the exact relationship between TDPE and an aggressive binding-time analysis. Although TDPE does not perform an explicit binding-time analysis, it does so implicitly. We find that using eta-expansion [20] enables the traditional approach to achieve the same results as TDPE. We illustrate this with an example drawn from an applied lambda calculus.

Consider the expression given by

$$E \quad \equiv \quad (\lambda\, f.f \mathbin{@} ((\mathit{if}\ \texttt{\#f}\ f\ g) \mathbin{@} 0)) \mathbin{@} (\lambda\, z.z). \qquad (1)$$

[1] in personal communication, March '98.

$$
\begin{array}{l}
\Downarrow \, : \, \mathit{Type} \to \mathit{TLT} \to \mathit{TLT} \\
\Uparrow \, : \, \mathit{Type} \to \mathit{TLT} \to \mathit{TLT}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{reify} & \Downarrow^{b} e & = e \\
 & \Downarrow^{t_1 \to t_2} e & = \underline{\lambda}\, x.\, \Downarrow^{t_2} (e \,\overline{@}\, (\Uparrow_{t_1} x)) \\
\mathit{reflect} & \Uparrow_{b} e & = e \\
 & \Uparrow_{t_1 \to t_2} e & = \overline{\lambda}\, v.\, \Uparrow_{t_2} (e \,\underline{@}\, \Downarrow^{t_1} v)
\end{array}
$$

Fig. 6. TDPE using two-level eta-expansion [15]

Assuming that g is known to be a function, but otherwise unknown (dynamic), we explore three different binding-time annotation schemes. But first, we rephrase TDPE so that it maps two-level expressions to two-level expressions.

4.1 TDPE and two-level eta-expansion

If we rewrite TDPE to a syntactic transformation on two-level terms as in Fig. 6, it turns out that this new specification is related to the implementation in Fig. 5 via the *cogen* function.

Lemma 1. *Suppose $e \in \mathit{TLT}$ with all annotations static and $\vdash_{bta} e : t$. Then*

1. $\vdash_{bta} \Downarrow^{t} e : \underline{t}$ *where the underlying type of $\underline{t}$ is identical to t, but with all annotations dynamic;*
2. $\downarrow^{t} \mathit{cogen}(e) = \mathit{cogen}(\Downarrow^{t} e)$.

4.2 Standard binding-time analysis

A standard binding-time analysis [27] (as implemented in PGG or in Similix) yields the two-level term

$$(\overline{\lambda}\, f.f \,\underline{@}\, ((\overline{\mathit{if}}\; \texttt{\#f}\;\; f\;\; g) \,\underline{@}\, 0)) \,\overline{@}\, (\underline{\lambda}\, z.z) \tag{2}$$

which statically reduces to

$$(\lambda\, z.z) \,@\, (g \,@\, 0).$$

This result is not satisfactory because the dynamic g pollutes the surrounding binding times.

4.3 Eta-expanding binding-time analysis

Since the type of the free variable g is known to be $A \to A$ for some base type A, a prephase of the binding-time analysis could exploit this knowledge and eta-expand g to yield

$$(\lambda\, f.f \,@\, ((\mathit{if}\; \texttt{\#f}\;\; f\;\; (\lambda\, w.g \,@\, w)) \,@\, 0)) \,@\, (\lambda\, z.z). \tag{3}$$

The same could be done with the type of the result. However, in the present example the result has type A, a base type.

Applying the same binding-time analysis as before to the term (3) yields

$$(\overline{\lambda}\, f.f \mathbin{\overline{@}} ((\overline{\mathit{if}}\ \texttt{\#f}\ f\ (\overline{\lambda}\, w.g \mathbin{\underline{@}} w)) \mathbin{\overline{@}} 0)) \mathbin{\overline{@}} (\overline{\lambda}\, z.z) \tag{4}$$

and static reduction of this term results in

$$g \mathbin{@} 0.$$

4.4 Type-directed partial evaluation

To simulate a binding-time analysis for TDPE, we first have to construct a completely static variant of E (as we only have the compiled program at our disposal) and close it over g:

$$E' \quad \equiv \quad \overline{\lambda}\, g.(\overline{\lambda}\, f.f \mathbin{\overline{@}} ((\overline{\mathit{if}}\ \texttt{\#f}\ f\ g) \mathbin{\overline{@}} 0)) \mathbin{\overline{@}} (\overline{\lambda}\, z.z) \tag{5}$$

The term E' has type $t = (A \to A) \to A$. Performing type-directed partial evaluation is the same as expanding $\Downarrow^t E'$ and then compiling and running the result. In this particular case, $\Downarrow^t [\,]$ corresponds to the context:

$$\underline{\lambda}\, g.[\,] \mathbin{\overline{@}} \overline{\lambda}\, w.g \mathbin{\underline{@}} w$$

The resulting term

$$\underline{\lambda}\, g.(\overline{\lambda}\, g.(\overline{\lambda}\, f.f \mathbin{\overline{@}} ((\overline{\mathit{if}}\ \texttt{\#f}\ f\ g) \mathbin{\overline{@}} 0)) \mathbin{\overline{@}} (\overline{\lambda}\, z.z)) \mathbin{\overline{@}} \overline{\lambda}\, w.g \mathbin{\underline{@}} w \tag{6}$$

is statically convertible with the term (4) constructed by binding-time analysis after eta expansion. One static reduction step brings us to

$$\underline{\lambda}\, g.(\overline{\lambda}\, f.f \mathbin{\overline{@}} ((\overline{\mathit{if}}\ \texttt{\#f}\ f\ (\overline{\lambda}\, w.g \mathbin{\underline{@}} w)) \mathbin{\overline{@}} 0)) \mathbin{\overline{@}} (\overline{\lambda}\, z.z) \tag{7}$$

which is identical to (4), up to the outermost abstraction of g. It statically reduces to

$$\lambda\, g.g \mathbin{@} 0. \tag{8}$$

It is easy to see that wrapping a free variable (or primitive operator) g whose type t is known into a bunch of eta-redexes corresponding to t enables the same reductions as TDPE.

Lemma 2. *Suppose $E[g]$ is an expression with free variable $g : t$. Then TDPE-style eta-expansion of $\lambda\, g.E[g]$ is statically convertible to $E[g']$ where each occurrence of g is replaced by $\Uparrow_t g$. That is*

$$\Downarrow^{t \to t'} \overline{\lambda}\, g.E[g] \text{ statically reduces to } \underline{\lambda}\, z. \Downarrow^{t'} E[\Uparrow_t z.$$

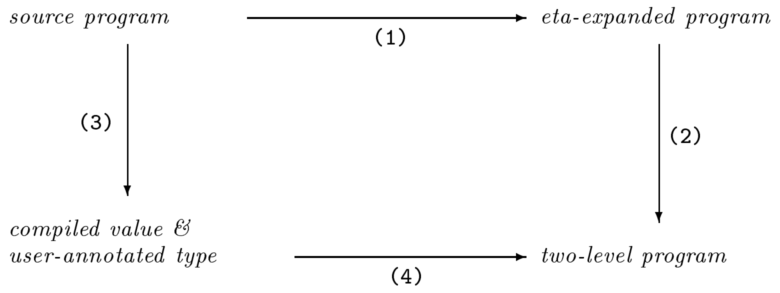

Fig. 7. TDPE and binding-time analysis

4.5 Assessment

Although it is possible to automate an eta-expanding prephase as in Sec. 4.3 to the binding-time analysis, current systems leave it to the programmer to perform the expansion manually as a binding-time improvement [19]. In contrast, TDPE *always* performs the eta-expansion because that is its fundamental working principle. If eta-expansion is not possible, e.g., if the top-level type has negative occurrences of recursive types, TDPE is not applicable, so manual workarounds have to be developed. In the traditional approach, eta-expansion is only required very rarely and it has to be done by hand, anyway.

A conclusion to draw from this example is that an eta-expanding binding-time analysis anticipates some static reduction steps that TDPE has to execute. This anticipation is possible because every binding-time analysis relies on a flow analysis. TDPE has to unveil this flow on the fly. However, its flow information is precise —in fact perfect— because it happens at run-time and exploits actual values (for example, TDPE ignores semantically dead code which may deteriorate the results of a static analysis), whereas a flow analysis can only provide conservative approximations.

The diagram in Fig. 7 summarizes this subsection. The arrow `(1)` signifies type-guided eta expansion, `(2)` is a binding-time analysis, `(3)` is compilation with user-annotated type information, and `(4)` is the reify/reflect phase of TDPE. The results of `(2)` and `(4)` are statically convertible.

4.6 Type dependencies

In our opinion, the strength of TDPE lies in applications where the type for residualization depends on the particular static input. Simple type specialization [17] is such an application. In those cases, the traditional approach is still viable in the presence of an eta-expanding binding-time analysis. However, it would be forbiddingly expensive because we would have to perform the binding-time analysis for each combination of the subject program and static input. In addition, we would have to run *cogen* and to compile the resulting generating extension.

On the other hand, many applications keep the type argument of *reify* fixed, among them compilation. Here, the extra power provided by TDPE cannot be exploited and it seems more advantageous to use a traditional cogen-based system because it provides additional features like memoization for free.

If the type argument t is fixed, self-application of TDPE [15, sec.2.4] can produce a customized eta-expander which is specialized with respect to t. Of course, any other specialization mechanism could do the job, too.

5 Quantitative comparison

In this section, we report on experiments that we conducted to examine the pragmatics of both systems more accurately. They are all concerned with compilation by specializing interpreters for various languages. These are:

Tiny [39] A simple imperative language with expressions, assignments, **if**, **while**, and sequence commands. We have examined one version in direct style and another in continuation-passing style. The same interpreters are considered in the TDPE papers [14, 15]. Since the interpreter already contained the proper combinators to handle dynamic recursion, no changes are required.

Mixwell [32] A first-order functional language with named procedures. The interpreter is written in direct style. As discussed in Sec. 3.7, a binding-time improvement is required to make the interpreter amenable to TDPE. The changes do not affect Mixwell's suitability for traditional specialization.

Mini-Scheme A large subset of Scheme. We have implemented this interpreter to be able to run realistic compilations. To this end, we have used the preprocessor of the PGG system to transform Scheme into Mini-Scheme. The implementation uses techniques similar to those developed for the Mixwell interpreter. Mini-Scheme is written in continuation-passing style.

5.1 Results of specialization

We have used both systems to specialize the above interpreters with respect to various programs. Since all programs that are amenable to TDPE can be processed by traditional specialization without using memoization, we switched this feature off in the PGG system [47]. In all cases we were able to obtain specialized programs that were identical up to renaming of bound variables.

- The Tiny interpreter in direct style is directly amenable to specialization with PGG without any modification. The residual programs are identical to those produced by TDPE.
- The Tiny interpreter in continuation-passing style needs some binding-time improvements for specialization with the PGG system in the form of eta expansions as described in Sec. 4.3.
- The Mixwell interpreter—in its TDPE-ready version—is directly amenable to PGG specialization and yields identical results from both specializers.

– The Mini-Scheme interpreter is equally amenable to TDPE and PGG and produces identical residual programs. In this case, the interpreter requires no eta expansion although it is written in continuation-passing style. This is due to the use of dynamic primitives in direct style.

5.2 Compile times

In the experiments, we have specialized the Mini-Scheme interpreter with respect to realistic programs taken from Andrew Wright's Scheme benchmark suite: **Boyer** a term-rewriting theorem prover, **Gpscheme** an implementation of genetic programming, and **Matrix** tests whether a random matrix $M \in \{+1, -1\}^{n \times n}$ is maximal under permutation and negation of rows and columns. **Append** is just a standard list appending program.

program	A: *TDPE*	B: *TDPE-cogen*	C: *PGG*	A/B	A/C	B/C
Append	0.16	0.15	0.12	1.07	1.33	1.25
Boyer	6.55	5.38	4.73	1.22	1.38	1.13
Gpscheme	10.13	8.85	8.87	1.14	1.14	0.99
Matrix	15.22	13.16	12.46	1.15	1.22	1.05
Compiler Generation		0.15	4.92			0.03

Table 1. Compile-time benchmarks (in sec.)

Table 1 shows compile times in seconds for these programs. The *TDPE* column indicates the times obtained with Danvy's TDPE system and with our own reimplementation of it. The *TDPE-cogen* column indicates the times obtained with self-applied TDPE [15, sec.2.4]. To obtain these figures, we used our reimplementation of TDPE. The *PGG* column lists the figures obtained using the PGG system. The remaining columns list ratios as indicated in their headers.

All experiments were performed on an IBM PowerPC Workstation with 128MB RAM using Scheme48, an implementation of Scheme based on a byte-code interpreter. Each column of data was obtained with a fresh Scheme48 session and an heap image of 12 MB. All figures are averages of 5 consecutively executed runs.

The *TDPE-cogen* column indicates that self-application does not play such a prominent role in achieving efficiency as in traditional partial evaluation. Although self-application removes the overhead of type interpretation, it only improves compile times by about 15%. Hence the interpretation of the type argument does not contribute essentially to the specialization time. In the traditional approach, self-application usually speeds up the specialization by a factor of 4–6.

Self-application at type t generates a term quite similar to $\Downarrow^t$ [ ]. It does not depend on a particular subject program, but is generally applicable to all programs of type t. This is in contrast to traditional generating extensions which are tailored to specialize one particular subject program.

We did not construct a TDPE compiler generator by hand. All it would do is construct the same term as the *TDPE-cogen*. It might do it faster, but we believe there is not much to improve on the compiler generation time in Table 1.

5.3 An analytical perspective

Let us consider the issue of compile time from an analytical perspective. Looking back at the core of both specialization algorithms in Figures 4 and 5, there is actually not so much difference between the way specialization proceeds.

Here is the case for PGG (see Fig. 4): For each lambda in the source program, the specializer produced by the PGG system processes one lambda and — in the dynamic case — it generates a fresh variable name, applies the function to it, and constructs a lambda expression. Static applications are just applications and a dynamic application just constructs an application expression.

For TDPE (see Fig. 5) it is quite similar: TDPE must process each lambda and each application in the source program. In addition to that, for each lambda that is deemed to be dynamic, it generates a fresh variable name, applies a function to it, and constructs a residual lambda. For each application that is deemed dynamic, it constructs a function that constructs the residual application.

From the above discussion, we extract the following facts:

- processing a static lambda, a dynamic lambda, a variable, or a static application involves exactly the same actions for cogen-based partial evaluation and TDPE;
- processing a dynamic application involves constructing the residual application in both cases, in addition TDPE has to construct a function to pass it to the compiled application.

The numbers in table 1, which are averages of several subsequent measurements, indicate that the extra lambda construction has no serious impact on compile times. Indeed, after removal of type interpretation, compilation with TDPE is about identical to compilation with PGG.

6 Related work and conclusion

Apart from Danvy's articles [14–16, 21] there are few investigations on the pragmatics of TDPE. In his M.Sc. thesis [41], Rhiger performs experiments with action semantics interpreters where he compares the space and time requirements in a TDPE system with those in Similix. He concludes that compilation with TDPE is considerably faster and uses less space.

In this paper, however, we compare TDPE with the technology used in the PGG system and obtain different results. This is because PGG is based on a hand written cogen which is faster than a self-applied partial evaluator.

We have found that TDPE delivers the same residual programs as traditional partial evaluation if the latter is coupled with an eta-expanding binding-time analysis. Even that was rarely necessary in the examples we considered. Disregarding program generator generation time, we illustrated that the time taken by a cogen-based partial evaluator is similar to the time taken by a TDPE system. However, TDPE allows for additional flexibility since it is possible to specify a different type for each static input.

The position of TDPE could be further strengthened if it were possible to integrate the unavoidable type inference phase into the system. The recent integration of TDPE into an ML programming system [3] is a promising step forward. It eliminates many of the problems in making a program "TDPE-ready". However, it is still insufficient for ambitious applications like simple type specialization [17] where the implementation of a type inference engine is left to the user.

The online extensions of TDPE [16] provide more flexibility for the user and achieve a certain degree of polyvariant specialization which is common in traditional specializers. They also enhance the transformational power of TDPE beyond the reach of traditional one-pass offline techniques.

References

1. Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).
2. Peter Holst Andersen and Carsten Kehler Holst. Termination analysis for offline partial evaluation of a higher order functional language. In Radhia Cousot, editor, *Proc. International Static Analysis Symposium, SAS'96*, volume 1145 of *Lecture Notes in Computer Science*, pages 67–82, Aachen, Germany, September 1996. Springer-Verlag.
3. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In *Proceedings of the ACM SIGPLAN Workshop on Types in Compilation (TIC'98)*, Lecture Notes in Computer Science, Kyoto, Japan, March 1998.
4. Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proc. of the 6th Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
5. Lars Birkedal and Morten Welinder. Hand-writing program generator generators. In Manuel V. Hermenegildo and Jaan Penjam, editors, *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '94)*, volume 844 of *Lecture Notes in Computer Science*, pages 198–214, Madrid, Spain, September 1994. Springer-Verlag.
6. Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.
7. Anders Bondorf. Improving binding times without explicit CPS-conversion. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, pages 1–10, San Francisco, California, USA, June 1992.
8. Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16(2):151–195, 1991.

9. Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. *Journal of Functional Programming*, 3(3):315–346, July 1993.
10. Mikhail A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.
11. Robert Cartwright and Mike Fagan. Soft typing. In *Proc. Conference on Programming Language Design and Implementation '91*, pages 278–292, Toronto, June 1991. ACM.
12. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proc. 20th Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
13. Charles Consel and Francois Noël. A general approach for run-time specialization and its application to C. In POPL1996 [40], pages 145–156.
14. Olivier Danvy. Pragmatics of type-directed partial evaluation. In Danvy et al. [18], pages 73–94.
15. Olivier Danvy. Type-directed partial evaluation. In POPL1996 [40], pages 242–257.
16. Olivier Danvy. Online type-directed partial evaluation. In *Proc. Third Fuji International Symposium on Functional and Logic Programming*, Kyoto, Japan, April 1998. World Scientific Press, Singapore.
17. Olivier Danvy. A simple solution to type specialization. Technical Report RS-98-1, BRICS, University of Aarhus, Denmark, January 1998. To appear in ICALP98.
18. Olivier Danvy, Robert Glück, and Peter Thiemann, editors. *Dagstuhl Seminar on Partial Evaluation 1996*, volume 1110 of *Lecture Notes in Computer Science*, Schloß Dagstuhl, Germany, February 1996. Springer-Verlag.
19. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, July 1995.
20. Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 18(6):730–751, November 1996.
21. Olivier Danvy and Rene Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Kuchen and Swierstra [33], pages 182–197.
22. Robert Glück and Jesper Jørgensen. An automatic program generator for multi-level specialization. *Lisp and Symbolic Computation*, 10(2):113–158, July 1997.
23. Robert Glück and Morten Heine Sørensen. A roadmap to metacomputation by supercompilation. In Danvy et al. [18], pages 137–160.
24. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–70, January 1991.
25. William L. Harrison and Samuel N. Kamin. Modular compilers based on monad transformers. In *IEEE International Conference on Computer Languages, ICCL 1998*, Chicago, USA, May 1998. IEEE Computer Society Press.
26. John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. *Mathematical Structures in Computer Science*, 7(5):507–542, 1997.
27. Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In Hughes [30], pages 448–472.
28. Fritz Henglein. Dynamic typing: Syntax and proof theory. *Science of Computer Programming*, 22:197–230, 1994.
29. Carsten Kehler Holst. Finiteness analysis. In Hughes [30], pages 473–495.
30. John Hughes, editor. *Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, Cambridge, MA, 1991. Springer-Verlag.

31. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.
32. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications*, pages 124–140, Dijon, France, 1985. Springer-Verlag. LNCS 202.
33. Herbert Kuchen and Doaitse Swierstra, editors. *International Symposium on Programming Languages, Implementations, Logics and Programs (PLILP '96)*, volume 1140 of *Lecture Notes in Computer Science*, Aachen, Germany, September 1996. Springer-Verlag.
34. John Launchbury and Carsten Kehler Holst. Handwriting cogen to avoid problems with static typing. In *Draft Proceedings, Fourth Annual Glasgow Workshop on Functional Programming*, pages 210–218, Skye, Scotland, 1991. Glasgow University.
35. Julia L. Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. In *Proc. Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 165–190, Sendai, Japan, September 1997. Springer-Verlag.
36. *Proc. 1990 ACM Conference on Lisp and Functional Programming*, Nice, France, 1990. ACM Press.
37. John C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.
38. Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.
39. Larry C. Paulson. Compiler generation from denotational semantics. In Bernhard Lorho, editor, *Methods and Tools for Compiler Construction*, pages 219–250. Cambridge University Press, 1984.
40. *Proc. 23rd Annual ACM Symposium on Principles of Programming Languages*, St. Petersburg, Fla., January 1996. ACM Press.
41. Morten Rhiger. A study in higher-order programming languages. Master's thesis, University of Aarhus, Aarhus, Denmark, December 1997.
42. Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In Charles Consel, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '97*, pages 22–35, Amsterdam, The Netherlands, June 1997. ACM Press.
43. Peter Thiemann. Cogen in six lines. In R. Kent Dybvig, editor, *Proc. International Conference on Functional Programming 1996*, pages 180–189, Philadelphia, PA, May 1996. ACM Press, New York.
44. Peter Thiemann. Correctness of a region-based binding-time analysis. In *Proc. Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference*, volume 6 of *Electronic Notes in Theoretical Computer Science*, page 26, Pittsburgh, PA, March 1997. Carnegie Mellon University, Elsevier Science BV. URL: http://www.elsevier.nl/locate/entcs/volume6.html.
45. Peter Thiemann. A unified framework for binding-time analysis. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 742–756, Lille, France, April 1997. Springer-Verlag.
46. Peter Thiemann. Aspects of the pgg system: Specialization for standard scheme. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation—Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, Lecture Notes in Computer Science. Springer-Verlag, 1998.

47. Peter Thiemann. *The PGG System—User Manual*. University of Nottingham, Nottingham, England, June 1998. Available from `ftp://ftp.informatik.uni-tuebingen.de/pub/PU/thiemann/software/pgg/`.