Deriving a Statically Typed Type-Directed Partial Evaluator

Morten Rhiger

BRICS [*]
Department of Computer Science
University of Aarhus [†]

Abstract

Type-directed partial evaluation was originally implemented in Scheme, a dynamically typed language. It has also been implemented in ML, a statically Hindley-Milner typed language. This note shows how the latter implementation can be derived from the former through a functional representation of inductively defined types.

1 Introduction

Type-directed partial evaluation is an approach to specializing a term written in a higher-order language. Such a higher-order term is specialized by normalizing it with respect to its type. Normalization is done by eta-expanding the term in a two-level lambda-calculus and statically beta-reducing the expanded term.

The two stages — eta-expansion and beta-reduction — share an intermediate result: a two-level term. We consider two versions of the type-directed partial evaluation algorithm:

(i) If the two-level term is represented as a value of an inductively defined data type then the algorithm can be implemented in any (Turing complete) language. In this case both eta-expansion and beta-reduction are implemented in this language.

(ii) If the algorithm is implemented in a higher-order language an alternative is to represent the two-level term as a mixture of object-language terms (dynamic parts) and implementation-language terms (static parts). In this case eta-expansion is implemented in the implementation language. It generates a two-level term whose implementation-language parts are beta-reduced by the native beta-reducer of the implementation language.

[*]Basic Research in Computer Science (`http://www.brics.dk`), Centre of the Danish National Research Foundation.

[†]Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark. E-mail: `mrhiger@brics.dk`

1.1 The Problem

Both of the versions of the algorithm are directed by the type of the term to be normalized. They are applied to a term and a representation of the type of the term. In (ii), if the type is given as an element of an inductively defined type then it is impossible to statically type-check the algorithm, and indeed this also has been implemented in a dynamically types language, Scheme [2, 4].

This note shows how to derive a statically typed analogue of (ii) from (i). It is obtained by Church-encoding types as higher-order values.

Andrzej Filinski was the first to use a Church-encoding of types for type-directed partial evaluation.

1.2 Related Work

The first statically typed version of type-directed partial evaluation is due to Andrzej Filinski in the spring of 1995. This unpublished work showed that type-directed partial evaluation could be implemented at all using a Hindley-Milner type system. The second one is due to Zhe Yang in the spring of 1996 [9]. Yang provides general methods for encoding type-indexed values in a Hindley-Milner typed language and applies them to type-directed partial evaluation. The present work was carried out in the fall of 1997 and, according to Olivier Danvy, it is the third independent implementation of type-directed partial evaluation in a Hindley-Milner typed language [7].

Kristoffer Rose implemented type-directed partial evaluation in Haskell in the spring of 1998 using type classes [8]. Haskell's type classes permit overloaded functions, i.e., functions that have several definitions, one for each type of argument. Type-directed partial evaluation fits exactly into this pattern.

In her M. Sc. thesis, Belmina Dzafic implements type-directed partial evaluation in Elf, a statically typed, constraint logical language (summer 1998) [5]. Furthermore, she proves the equivalence of the dynamically typed and the statically typed versions of type-directed partial evaluation. Earlier on, Catarina Coquand stated and proved the correctness of the type-directed partial evaluation algorithm using the proof editor Alf [3].

1.3 The Derivation

Type-directed partial evaluation is given by $\downarrow$ (reify) and $\uparrow$ (reflect) in Figure 1. Based on this algorithm we derive a

$$
\begin{array}{lrcl}
\text{(types)} & t & ::= & \mathrm{b} \mid t_1 \rightarrow t_2 \\
\\
\text{(reify)} & \downarrow^{\mathrm{b}} v & = & v \\
 & \downarrow^{t_1 \rightarrow t_2} v & = & \underline{\lambda} x.\ \downarrow^{t_2} (v @ (\uparrow_{t_1}\ x)) \\
 & & & \text{where } x \text{ is fresh} \\
\\
\text{(reflect)} & \uparrow_{\mathrm{b}} e & = & e \\
 & \uparrow_{t_1 \rightarrow t_2} e & = & \lambda x.\ \uparrow_{t_2} (e \underline{@} (\downarrow^{t_1}\ x)) \\
\\
 & \mathrm{tdpe}\, t\, v & = & \downarrow^{t} v
\end{array}
$$

Figure 1: Type-directed partial evaluation

statically typed, type-directed partial evaluator analogous to (ii) in the following steps:

- (Section 2.1) Starting from (i), we represent types as a datatype `Type` and representing both static and dynamic terms as a datatype `Term` we translate the $\downarrow$ and $\uparrow$ of Figure 1 into into `reify` and `reflect`. This solution is interpretive in that it uses an explicit static beta-reducer.
- (Section 2.2) We change the representation of static terms from elements of the datatype `Term` into higher-order values and replace the explicit beta reducer by the native beta reducer of the implementation language. This solution is not interpretive but it is also not statically typeable.
- (Section 3.1) We observe an invariant property: `reify` is applied only to types occurring covariantly in the source type, and `reflect` only to types appearing contravariantly in the source type. We encode this distinction in the datatype `Type`. This solution is not statically typeable but by changing the representation of types from the datatype `Type` to higher-order values we obtain a statically typed solution which is still not interpretive: it uses the native (implicit) beta-reducer to statically reduce the two-level term.

Using the implicit static beta-reducer or an explicit one is independent of the representation of types. However, using the implicit static beta-reducer together with a datatype representation of types does not yield a solution that is statically typeable in a Hindley-Milner typing system.

1.4 Overview

We consider four successive implementations of type-directed partial evaluation in this note.

In Section 2, object-language types are represented by an inductively defined type. Terms are represented by values of an inductively defined type in Section 2.1 and by a mixture of higher-order values and terms in Section 2.2.

In Section 3, types are represented by higher-order values. The main result of this note is in Section 3.1 where terms are represented by higher-order values. For completeness we take a step backwards in Section 3.2 where terms are represented by values of an inductively defined data type.

In Section 3.3 we briefly consider the efficencies of the programs in the note. Section 4 concludes and in appendix A we present two extensions over the statically typed algorithm.

All programs in this note are written in a functional notation à la Haskell [6]. We defer the problem of generating fresh identifiers [7].

2 Inductively Defined Representation of Types

First we consider two implementations of type-directed partial evaluation that use a representation of types as elements of the following inductively defined type.

```
data Type =  Base | Func Type Type
```

The two implementations differ in the way the static parts of two-level terms are represented.

2.1 Inductively Defined Representation of Terms

In this approach, of which only an outline is given here, both the static and the dynamic parts of terms are represented by an inductively defined type.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term
```

The static syntax constructors are prefixed with an "`S`" and the dynamic syntax constructors are prefixed with a "`D`". Constants are both static and dynamic.

The algorithm proceeds in two stages: First, given a completely static term and its type, a fully eta-expanded two-level term is constructed using `reify` and `reflect` below. Second, the static parts of the term are beta-reduced (the beta-reducer is omitted here).

```
reify, reflect         :: Type -> Term -> Term

reify   Base         v =  v
reify   (Func t1 t2) v =
    DLam x (reify t2 (SApp v (reflect t1 (DVar x))))
    where x = fresh "x"

reflect Base         v =  v
reflect (Func t1 t2) v =
    SLam x (reflect t2 (DApp v (reify t1 (SVar x))))
    where x = fresh "x"
```

```
etaExpand t v      =  reify t v
staticBetaReduce v =  ...

tdpe               :: Type -> Term -> Term
tdpe t v           =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable* in a Hindley-Milner typing system and can be implemented in any language with inductively defined types, regardless of the typing discipline. The implementation-language type of the output of `reify` and `reflect` does not depend on the representation of the object-language type (a value of type `Type`).

The explicit use of a beta-reducer is undesirable since it embeds the lambda calculus into the implementing language via an interpreter. This is inefficient and the question arises whether we could not implement the algorithm in a higher-order language using the underlying beta-reduction mechanism of this implementation language.

2.2 Higher-Order Representation of Terms

The following approach uses the beta-reduction mechanism of the higher-order implementation language.

Types are represented by the same type as above. Two-level terms are represented by a mixture of implementation-language terms (values) and object-language terms.

```
reify   Base         v =  v
reify   (Func t1 t2) v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect Base         v =  v
reflect (Func t1 t2) v =
    \x -> reflect t2 (DApp v (reify t1 x))

etaExpand  t v  = reify t v

tdpe t v        = etaExpand t v
```

This program is *not statically typeable* in a Hindley-Milner typing system: The implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (a value of type `Type`).

Short of dependent types, at compile time, there is not enough information available so that the type-checker can accept the program. The above program corresponds to the original implementation of type-directed partial evaluation in Scheme [4].

3 Higher-Order Representation of Types

Observe that `reify` is always applied to types that occur *covariantly* in the source type (a value of type `Type`) and that `reflect` is always applied to types that occur *contravariantly* in the source type. We make this explicit by distinguishing between covariant occurrences (postfixed by "P" for positive) and contravariant occurrences (postfixed by "N" for negative):

```
data TypeP =  BaseP | FuncP TypeN TypeP
data TypeN =  BaseN | FuncN TypeP TypeN
type Type  =  TypeP
```

3.1 Higher-Order Representation of Terms

Now `reify` and `reflect` are

```
reify BaseP           v =  v
reify (FuncP t1 t2)   v =
    DLam x (reify t2 (v (reflect t1 (DVar x))))
    where x = fresh "x"
reflect BaseN         v =  v
reflect (FuncN t1 t2) v =
    \x -> reflect t2 (DApp v (reify t1 x))
```

This program is *not statically typeable* in a Hindley-Milner typing system. Again, the implementation-language type of the output of `reify` and `reflect` depends on the representation of the object-language type (values of types `TypeP` and `TypeN`).

In order to obtain a statically typeable program we apply the following change: instead of representing a positively occuring type t as a `TypeP` we represent it as a value equal to `(reify` t`)` and instead of representing a negatively occuring type t as a `TypeN` we represent it as a value equal to `(reflect` t`)`.

```
baseP, baseN     :: a -> a
funcP ::
    (Term -> a) -> (b -> Term) -> (a -> b) -> Term
funcN ::
    (a -> Term) -> (Term -> b) -> Term -> (a -> b)

baseP       v    =  v
funcP t1 t2 v    =  DLam x (t2 (v (t1 (DVar x))))
                    where x = fresh "x"
baseN       v    =  v
funcN t1 t2 v    =  \x -> t2 (DApp v (t1 x))

etaExpand t v    =  t v

tdpe             :: (a -> b) -> a -> b
tdpe t v         =  etaExpand t v
```

This program is *statically typeable* in a Hindley-Milner typing system. The implementation-language type of `tdpe` does not depend on the representation of the object-language type, but on the type of the object-language type.

Thus, even without dependent types, the type-checker has enough information to instantiate the polymorphic type of `tdpe`. This is the main result of this note.

For example, the type b (a base type) is represented by `Base` of type `Type` in Section 2.1. In the current section it is represented by `baseP` (i.e., the identity function) of type `a -> a`. Filinski and Yang's representation of this type is the pair of functions $(\downarrow^{\mathrm{b}}, \uparrow_{\mathrm{b}})$, i.e., a pair of identity functions. (See section A.2).

The type $\mathrm{b} \rightarrow \mathrm{b}$ is represented by `Func Base Base` of type `Type` in Section 2.1. In the current section it is represented by `funcP baseN baseP` of type `(Term -> Term) -> Term`. Filinski and Yang's representation of this type is the pair of functions $(\downarrow^{\mathrm{b}\rightarrow\mathrm{b}}, \uparrow_{\mathrm{b}\rightarrow\mathrm{b}})$.

As an example, let's specialize some terms that contains static redeces using the result of this section:

```
> :t tdpe baseP
tdpe baseP :: a -> a
> :type tdpe (funcP baseN baseP)
tdpe (funcP baseN baseP) :: (Term -> Term) -> Term
> tdpe baseP ((\x -> x) (Num 42))
Num 42
> tdpe (funcP baseN baseP) ((\x -> \y -> x) (Num 42))
DLam "x0" (Num 42)
>
```

3.2 Inductively Defined Representation of Terms

For the record, let us repeat the solution above using a "traditional" inductively defined representation of terms. To this end we use again the type of terms.

```
type Id   =  [Char]
data Term =  Num Int | Str String
          |  SVar Id | SLam Id Term | SApp Term Term
          |  DVar Id | DLam Id Term | DApp Term Term

baseP       v =  v
funcP t1 t2 v =  DLam x (t2 (SApp v (t1 (DVar x))))
                 where x = fresh "x"
baseN       v =  v
funcN t1 t2 v =  SApp x (t2 (DApp v (t1 (SVar x))))
                 where x = fresh "x"

etaExpand t v        =  t v
staticBetaReduce v   =  ...

tdpe     :: Type -> Term -> Term
tdpe t v =  staticBetaReduce (etaExpand t v)
```

This program is *statically typeable*. It also requires explicit static beta-reduction (which is omitted here).

3.3 Pragmatics

We have compared the performance of the three statically typed solution in ML. We used a simple-minded, hand-coded static beta-reducer for the programs in Section 2.1 and Section 3.2, and the native beta-reducer of ML for the program in Section 3.1. The hand-coded reducer uses the same reduction strategy as the native reducer of ML.

Specializing the power function with respect to a static exponent of value 12 is about 9 times faster using the solution of Section 3.1 than using the solutions of sections 2.1 and 3.2. Specializing $(S\,K)\,K$ at type $\mathrm{b} \to \mathrm{b}$ is about 4 times faster using the solution of Section 3.1 than using the solutions of sections 2.1 and 3.2. These results confirm Berger, Eberl, and Schwichtenberg's empirical observations [1].

There do not appear to be any perceptible difference between the two solutions that use the hand-coded static reducer (Section 2.1 and Section 3.2).

4 Conclusion

Being directed by the *type* of a term, the type-directed partial evaluation algorithm requires a way of representing types. In dynamically typed languages an inductively defined sum over the different kind of types (base types, product types, function types, etc.) suffices. In statically typed languages with a Hindley-Milner typing system this does not work: the *type* of the algorithm depends on the *value* of the representation of the type.

The solution is to represent types as higher-order polymorphic functions. This works since the *type* of the algorithm thus depends on the implementation-language *type* of the representation of the object-language type.

Our work suggests to view the higher-order encoding as a functional representation of types, specialised to the purpose of being deconstructed by reify and reflect.

Acknowledgements

Thanks are due to Olivier Danvy and the anonymous referees for commenting earlier versions of this paper. Also many thanks to Kristoffer Rose for shepherding this paper.

A Extending the Statically Typed Algorithm

Consider again

```
tdpe baseP                :: a -> a
tdpe (funcP baseN baseP) :: (Term -> Term) -> Term
```

This indicates that constants of base type must be coerced to dynamic values in the source programs. For example, to obtain something of type `Term` we must coerce the integer 42 into a `Term` in

```
> tdpe baseP (Num 42)
Num 42
> tdpe (funcP baseN baseP) (\x -> (Num 42))
DLam "x0" (Num 42)
>
```

Furthermore, we must explicitly indicate the variance of types (by the postfix "P" or "N"). Both shortcomings are alleviated below.

A.1 The Need for Coercing Base Values

At covariant base types the explicit coercion of values of base type can be removed by distinguishing the base types. We introduce a more specific version of `baseP` for each base type.

```
numP    v =  Num v
strP    v =  Str v

tdpe numP               :: Int -> Term
tdpe strP               :: String -> Term
tdpe (funcP baseP numP) :: (Term -> Int) -> Term
```

These new functions will reify a static value into its dynamic counterpart. Static values of base type, such as integers and strings, are represented uniquely and these values are used directly when constructing the dynamic term. A similar solution does not work at contravariant base type since dynamic values of base type can be any dynamic term.

```
> tdpe (funcP baseN numP) (\x -> 42)
DLam "x0" (Num 42)
> tdpe (funcP baseN strP) (\x -> "fortytwo")
DLam "x0" (Str "fortytwo")
>
```

A.2 The Need for Specifying the Variance

The other shortcoming of the implementation — the explicit distinction between covariant and contravariant types — can also be alleviated. Instead of representing a type in two parts (i.e., a covariant part and a contravariant part as above) we can merge the two parts into a pair that represents the type, obtaining Filinski and Yang's solution [9].

```
base       :: (a -> a, b -> b)
func       :: (a -> Term, Term -> b) ->
                (c -> Term, Term -> d) ->
                  ((b -> c) -> Term, Term -> (a -> d))

base       = (reify, reflect)
    where reify   v = v
          reflect v = v
func t1 t2 = (reify, reflect)
    where reify   v =
              DLam x (fst t2 (v (snd t1 (DVar x))))
              where x = fresh "x"
          reflect v =
              \x -> (snd t2 (DApp v (fst t1 x))))

etaExpand t v  =  (fst t) v

tdpe           :: (a -> b,c) -> a -> b
tdpe t v       =  etaExpand t v
```

Using this implementation we can specialise terms without specifying the covariance and contravariance of the type involved.

```
> tdpe (func (func base base) base) (\f -> f (Num 8))
DLam "x0" (DApp (DVar "x0") (Num 8))
>
```

References

[1] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. October 1998. Draft. http://www.mathematik.uni-muenchen.de/~schwicht/n8kurz.ps.Z

[2] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

[3] Catarina Coquand. From semantics to rules: A machine assisted analysis. In Egon Börger, Yuri Gurevich, and Karl Meinke, editors, *Proceedings of CSL'93*, number 832 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[4] 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.

[5] Belmina Dzafic. Formalizing program transformations. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1998.

[6] Joseph H. Fasel, Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). Haskell special issue. *SIGPLAN Notices*, 27(5), May 1992.

[7] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[8] Kristoffer Rose. Type-directed partial evaluation using type classes. In Olivier Danvy and Peter Dybjer, editors, *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98,* (Chalmers, Sweden, May 8–9, 1998), number NS-98-1 in BRICS Note Series, Department of Computer Science, University of Aarhus, May 1998.

[9] 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*, Baltimore, Maryland, September 1998. ACM Press. Extended version available as the technical report BRICS RS-98-9.