A Simple Take on Typed Abstract Syntax in Haskell-like Languages

Olivier Danvy
Morten Rhiger

BRICS Report Series RS-00-34

ISSN 0909-0878 December 2000

A Simple Take on Typed Abstract Syntax in Haskell-like Languages (Extended version) [*]

Olivier Danvy and Morten Rhiger

BRICS [†]

Department of Computer Science

University of Aarhus [‡]

December, 2000

Abstract

We present a simple way to program typed abstract syntax in a language following a Hindley-Milner typing discipline, such as Haskell and ML, and we apply it to automate two proofs about normalization functions as embodied in type-directed partial evaluation for the simply typed lambda calculus: normalization functions (1) preserve types and (2) yield long beta-eta normal forms.

Keywords: Type-directed partial evaluation, normalization functions, simply typed lambda-calculus, higher-order abstract syntax, Haskell.

[*]To appear in the proceedings of the Fifth International Symposium on Functional and Logic Programming (FLOPS 2001), March 7-9, 2001, Waseda University, Tokyo, Japan.

[†]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: {`danvy,mrhiger`}`@brics.dk`
Home pages: `http://www.brics.dk/~`{`danvy,mrhiger`}

Contents

List of Figures

1 Introduction

Programs (implemented in a *meta language*) that manipulate programs (implemented in an *object language*) need a representation of the manipulated programs. Examples of such programs include interpreters, compilers, partial evaluators, and logical frameworks.

When the meta language is a functional language with a Hindley-Milner type system, such as Haskell [2] or ML [7], a data type is usually chosen to represent object programs. In functional languages, data types are instrumental in representing sum types and inductive types, both of which are needed to represent even the simplest programs such as arithmetic expressions.

However, the *object-language types* of object-language terms represented by data types cannot be inferred from the representation if the meta language does not provide dependent types. Hence, regardless of any typing discipline in the object language, when the meta language follows a Hindley-Milner type discipline, it cannot prevent the construction of object-language terms that are untyped, and correspondingly, it cannot report the types of object-language terms that are well-typed. This typeless situation is familiar to anyone who has represented λ-terms using a data type in an Haskell-like language.

In this article we consider a simple way of representing monomorphically typed λ-terms in an Haskell-like language. We describe a typeful representation of terms that prevents one from constructing untyped object-language terms in the meta language and that makes the type system of the meta language report the types of well-typed object-language terms.

We apply this typeful representation to *type-directed partial evaluation* [1, 3], using Haskell [9]. In Haskell, the object language of type-directed partial evaluation is a subset of the meta language, namely the monomorphically typed λ-calculus. Type-directed partial evaluation is an implementation of *normalization functions.* As such, it maps a meta-language value that is simply typed into a (textual) representation of its *long beta-eta normal form.*

All previous implementations of type-directed partial evaluation in Haskell-like languages have the type $t \to \mathsf{Term}$, for some t and where Term denotes the typeless representation of object programs. This type does not express that the output of type-directed partial evaluation is a representation of an object of the same type as the input. In contrast, our implementation has the more expressive type $t \to \mathsf{Exp}(t)$, where Exp denotes our typeful representation of object programs. This type proves that type-directed partial evaluation preserves types. Furthermore, using the same technique, we also prove that the output of type-directed partial evaluation is indeed in long beta-eta normal form.

The rest of this article is organized as follows. In Section 2 we review a traditional, typeless data-type representation of λ-terms in Haskell. In Section 3, we review higher-order abstract syntax, which is a stepping stone towards our typeful representation. Section 4 presents our main result, namely an extension of higher-order abstract syntax that only allows well-typed object-language terms to be constructed. In Section 5, we review type-directed partial evaluation, which is our chosen domain of application. Section 6 presents our first application, namely an implementation of type-directed partial evaluation preserving types. Section 7 presents our second application, namely another implementation of type-directed partial evaluation demonstrating that it produces long beta-eta normal forms. Section 8 concludes.

2 Typeless first-order abstract syntax

We consider the simply typed λ-calculus with integer constants, variables, applications, and function abstractions:

$$\begin{array}{llll} \text{(Types)} & t & ::= & \alpha \mid \mathsf{int} \mid t_1 \to t_2 \\ \text{(Terms)} & e & ::= & i \mid x \mid e_0\, e_1 \mid \lambda x.e \end{array}$$

Other base types (booleans, reals, etc.) and other type constructors (products, sums, lists, etc.) are easy to add. So our object language is the λ-calculus.

Our meta language is Haskell. We use the following data type to represents λ-terms. Its constructors are: integers (`INT`), variables (`VAR`), applications (`APP`), and functional abstractions (`LAM`).

```
data Term = INT Int
          | VAR String
          | APP Term Term
          | LAM String Term
```

Object-language terms are constructed in Haskell using the translation below. Note that the type of $\lceil e \rceil_0$ is Term regardless of the type of e in the λ-calculus.

$$\begin{array}{rcl} \lceil i \rceil_0 & = & \texttt{INT}\ i \\ \lceil x \rceil_0 & = & \texttt{VAR "}x\texttt{"} \\ \lceil e_0\, e_1 \rceil_0 & = & \texttt{APP}\ \lceil e_0 \rceil_0\ \lceil e_1 \rceil_0 \\ \lceil \lambda x.e \rceil_0 & = & \texttt{LAM "}x\texttt{"}\ \lceil e \rceil_0 \end{array}$$

The constructors of the data type are typed in Haskell: The term `INT 9` is valid whereas `INT "a"` is not. However, Haskell knows nothing of the λ-terms we wish to represent. In other words, the translation $\lceil \cdot \rceil_0$ is not surjective: Some well-typed *encodings* of object-language terms do not correspond to any legal object-language term. For example, the term `APP (INT 1) (LAM "x" (VAR "x"))` has type Term in Haskell, even though it represents the term $1(\lambda x.x)$ which has no type in the λ-calculus.

The fact that we can represent untyped λ-terms is not a shortcoming of the meta language. One might want to represent programs in an untyped object language like Scheme [6] or even structures for which no notion of type exists.

3 Typeless higher-order abstract syntax

To the data type Term we add an interface using *higher-order abstract syntax* [8]. In higher-order abstract syntax, object-language variables and bindings

```
module TypelessExp(int, app, lam, Exp) where

  data Term = INT Int | VAR String | APP Term Term | LAM String Term

  type Exp = Int -> Term

  int i     j = INT i
  app e0 e1 j = APP (e0 j) (e1 j)
  lam f     j = LAM v (f (\_ -> VAR v) (j + 1))
                where v = "x" ++ show j
```

Figure 1: Typeless higher-order abstract syntax in Haskell

are represented by meta-language variables and bindings. The interface to the data type Term is shown in Figure 1.

The interface consists of syntax constructors for integers, applications, and abstractions. There is no constructor for variables. Instead, fresh variable names are generated and passed to the higher-order representation of abstractions. A λ-expression is represented by a function accepting the next available fresh-variable name, using de Bruijn levels.

Object-language terms are constructed in the meta language using the following translation. Note again that the type of $\lceil e \rceil_1$ is Term regardless of the type of e in the λ-calculus.

$$\begin{array}{rcl} \lceil i \rceil_1 &=& \texttt{int}\ i \\ \lceil x \rceil_1 &=& x \\ \lceil e_0\, e_1 \rceil_1 &=& \texttt{app}\ \lceil e_0 \rceil_1\ \lceil e_1 \rceil_1 \\ \lceil \lambda x.e \rceil_1 &=& \texttt{lam (\textbackslash}x\ \texttt{->}\ \lceil e \rceil_1\texttt{)} \end{array}$$

This translation is also not surjective in the sense outlined in Section 2. Indeed, the types of the three higher-order constructors in Haskell still allow untypable λ-terms to be constructed. These three constructors are typed as follows.

$$\begin{array}{rcl} \texttt{int} &::& \mathsf{Int} \to \mathsf{Exp} \\ \texttt{app} &::& \mathsf{Exp} \to (\mathsf{Exp} \to \mathsf{Exp}) \\ \texttt{lam} &::& (\mathsf{Exp} \to \mathsf{Exp}) \to \mathsf{Exp} \end{array}$$

Therefore, the term `app (int 1) (lam (\x -> x))` still has a type in Haskell, namely Exp.

4 Typeful higher-order abstract syntax

Let us restrict the three higher-order constructors above to only yield well-typed terms. To this end, we make the following observations about constructing well-typed terms.

```
module TypefulExp (int, app, lam, Exp) where

  data Term = INT Int | VAR String | APP Term Term | LAM String Term

  data Exp t = EXP (Int -> Term)

  int :: Int -> Exp Int
  app :: Exp (a -> b) -> Exp a -> Exp b
  lam :: (Exp a -> Exp b) -> Exp (a -> b)

  int i                 = EXP (\x -> INT i)
  app (EXP e0) (EXP e1) = EXP (\x -> APP (e0 x) (e1 x))
  lam f                 = EXP (\x -> let v     = "x" ++ show x
                                             EXP b = f (EXP (\_ -> VAR v))
                                         in  LAM v (b (x + 1)))
```

Figure 2: Typeful higher-order abstract syntax in Haskell

- The constructor `int` produces a term of object-language type Int.
- The first argument to `app` is a term of object-language type $\alpha \to \beta$, the second argument is a term of object-language type α, and `app` produces a term of object-language type β.
- The argument to `lam` must be a function mapping a term of object-language type α into a term of object-language type β, and `lam` produces a term of object-language type $\alpha \to \beta$.

These observations suggest that the (polymorphic) types of the three constructors actually could reflect the object-language types. We thus parameterize the type Exp with the object-language type and we restrict the types of the constructors according to these observations. In Haskell we implement the new type constructor as a data type, not just as an alias for Int → Term as in Figure 1. In this way the internal representation is hidden. The result is shown in Figure 2. The three constructors are typed as follows.

$$\begin{array}{lcl} \texttt{int} & :: & \mathsf{Int} \to \mathsf{Exp}(\mathsf{Int}) \\ \texttt{app} & :: & \mathsf{Exp}(\alpha \to \beta) \to (\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\beta)) \\ \texttt{lam} & :: & (\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\beta)) \to \mathsf{Exp}(\alpha \to \beta) \end{array}$$

The translation from object-language terms to meta-language terms is the same as the one for the typeless higher-order abstract syntax. However, unlike for the typeless version, if e is an (object-language) term of type t then the (meta-language) type of $\lceil e \rceil_1$ is $\mathsf{Exp}(t)$.

As an example, consider the λ-term $\lambda f.f(1)$ of type $(\mathsf{Int} \to \alpha) \to \alpha$. It is encoded in Haskell by $\lceil \lambda f.f(1) \rceil_1$ = `lam (\f -> app f (int 1))` of type $\mathsf{Exp}((\mathsf{Int} \to \alpha) \to \alpha)$. Now consider the λ-term $1(\lambda x.x)$ which is not well-typed

```
module TypelessTdpe where

  import TypelessExp  -- from Figure 1

  data Reify_Reflect(a) =
    RR { reify   :: a -> Exp,
         reflect :: Exp -> a }

  rra =                -- atomic types
    RR { reify   = \x -> x,
         reflect = \x -> x }

  rrf (t1, t2) =       -- function types
    RR { reify   = \v -> lam (\x -> reify t2 (v (reflect t1 x))),
         reflect = \e -> \x -> reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 3: A typeless implementation of type-directed partial evaluation

in the λ-calculus. It is encoded by $\lceil 1(\lambda x.x) \rceil_1 =$ `app 1 (lam (\x -> x))` which is rejected by Haskell.

In the remaining sections, we apply typeful abstract syntax to type-directed partial evaluation.

5 Type-directed partial evaluation

The goal of *partial evaluation* [5] is to specialize a program p of type $t_1 \to t_2 \to t_3$ to a fixed first argument v of type t_1. The result is a *residual* program p_v that satisfies $p_v(w) = p(v)(w)$ for all w of type t_2, if both expressions terminate. The motivation for partial evaluation is that running $p_v(w)$ is more efficient than running $p(v)(w)$.

In *type-directed partial evaluation* [1, 3, 9, 10], specialization is achieved by normalization. For simply typed λ-terms, the partial application $p(v)$ is residualized into (the text of) a program p_v in *long beta-eta normal form.* That is, the residual program contains no beta-redexes and it is fully eta-expanded with respect to its type.

5.1 Type-directed partial evaluation in Haskell

Figure 3 displays a typeless implementation of type-directed partial evaluation for the simply typed λ-calculus in Haskell. To normalize a polymorphic value v of type t, one applies the main function `normalize` to the value, v, and a *representation* of the type, $|t|$, defined as follows.

$$\begin{array}{rcl} |\alpha| & = & \texttt{rra} \\ |t_1 \to t_2| & = & \texttt{rrf(}|t_1|\texttt{, }|t_2|\texttt{)} \end{array}$$

To analyze the type of the representations of types, we first define the *instance* of a type as follows.

$$\begin{array}{rcl} [\alpha]_0 & = & \mathsf{Exp} \\ [t_1 \to t_2]_0 & = & [t_1]_0 \to [t_2]_0 \end{array}$$

Then, for any type t, the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_0)$. Haskell infers the following type for the main function.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}$$

This type shows that `normalize` maps a α-typed input value into an Exp-typed output value, i.e., a term. This type, however, does not show that the input (meta-language) value and the output (object-language) term have the same type. In Section 6, we show that type-directed partial evaluation is type-preserving, and in Section 7, we show that the output term is in normal form.

5.2 Example: Church numerals, typelessly

As an example, we apply type-directed partial evaluation to specialize the addition of two Church numerals with respect to one argument. The Church numeral zero, the successor function, and addition are defined as follows.

```
zero    :: (a -> a) -> a -> a
zero    = \s -> \z -> z
suc n   = \s -> \z -> s (n s z)
add m n = \s -> \z -> m s (n s z)
```

5.2.1 Specializing `add` with respect to 0

We specialize the addition function with respect to the Church numeral 0 by normalizing the partial application `add zero`. This expression has the following type.

$$t_{\text{add}} = ((\alpha \to \alpha) \to \beta \to \alpha) \to (\alpha \to \alpha) \to \beta \to \alpha$$

This type is represented in Haskell as follows.

$$\begin{array}{l} |t_{\text{add}}| = \texttt{rrf(rrf(rrf(rra, rra), rrf(rra, rra)),} \\ \qquad\qquad\quad \texttt{rrf(rrf(rra, rra), rrf(rra, rra)))} \end{array}$$

Thus, evaluating the Haskell expression

$$\texttt{normalize}\ |t_{\text{add}}|\ \texttt{(add zero) 37}$$

(taking 37, for example, as the first de Bruijn level) yields a representation of the following residual term.

$$\lambda x_{37}.\lambda x_{38}.\lambda x_{39}.x_{37}(\lambda x_{40}.x_{38}\ x_{40})x_{39}$$

For readability, let us rename this residual term:

$$\lambda n.\lambda s.\lambda z.n(\lambda n'.s\ n')z$$

This term is the (η-expanded) identity function over Church numerals, reflecting that 0 is neutral for addition.

Haskell infers the following type of the expression `normalize` $|t_{\mathrm{add}}|$.

$$(((t' \to t') \to t' \to t') \to (t' \to t') \to t' \to t') \to t', \quad \text{where } t' = \mathsf{Int} \to \mathsf{Term}$$

This type does not express any relationship between the type of the input term and the type of the residual term.

5.2.2 Specializing `add` with respect to 5

We specialize the addition function with respect to the Church numeral 5 by normalizing the partial application `add five`, where `five` is defined as follows.

```
five    = suc (suc (suc (suc (suc zero))))
```

The expression `add five` also has the type t_{add}. Thus, evaluating the Haskell expression

`normalize` $|t_{\mathrm{add}}|$ `(add five) 57`

(taking 57 this time as the first de Bruijn level) yields a representation of the following residual term.

$$\lambda x_{57}.\lambda x_{58}.\lambda x_{59}.x_{58}(x_{58}(x_{58}(x_{58}(x_{58}(x_{57}(\lambda x_{60}.x_{58}\ x_{60})x_{59})))))$$

For readability, let us rename this residual term:

$$\lambda n.\lambda s.\lambda z.s(s(s(s(s(n(\lambda n'.s\ n')z)))))$$

In this term, the successor function is applied five times, reflecting that the addition function has been specialized with respect to five.

6 Application 1: type preservation

In this section, we use the type inferencer of Haskell as a theorem prover to show that type-directed partial evaluation preserves types. To this end, we implement type-directed partial evaluation using typed abstract syntax.

```
module TypefulExpCoerce (int, app, lam, coerce, uncoerce, Exp) where

  [...]

  coerce   :: Exp a -> Exp (Exp a)
  uncoerce :: Exp (Exp a) -> Exp a

  coerce   (EXP f) = EXP f
  uncoerce (EXP f) = EXP f
```

Figure 4: Typeful higher-order abstract syntax with coercions for atomic types

6.1 Typeful type-directed partial evaluation (first variant)

We want the type of `normalize` to be $\mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}(\alpha)$. As a first step to achieve this more expressive type, we shift to the typeful representation of terms from Figure 2. The parameterized type constructor $\mathsf{Exp}(\alpha)$ replaces the type Exp. Thus, we change the data type $\mathsf{Reify\_Reflect}(\alpha)$ from Figure 3 to the following.

```
data Reify_Reflect a =
  RR { reify   :: a -> Exp a,
       reflect :: Exp a -> a }
```

This change, however, makes the standard definition of `rra` untypable: The identity function does not have type $\alpha \to \mathsf{Exp}(\alpha)$ (or $\mathsf{Exp}(\alpha) \to \alpha$ for that matter). We solve this problem by introducing two identity functions in the module of typed terms.

$$\begin{array}{rcl} \texttt{coerce} & :: & \mathsf{Exp}(\alpha) \to \mathsf{Exp}(\mathsf{Exp}(\alpha)) \\ \texttt{uncoerce} & :: & \mathsf{Exp}(\mathsf{Exp}(\alpha)) \to \mathsf{Exp}(\alpha) \end{array}$$

At first it might seem that a function of type $\mathsf{Exp}(\alpha) \to \mathsf{Exp}(\mathsf{Exp}(\alpha))$ cannot be the identity. However, internally $\mathsf{Exp}(t)$ is an alias for $\mathsf{Int} \to \mathsf{Term}$, thus discarding t, so in effect we are looking at two identity functions of type $(\mathsf{Int} \to \mathsf{Term}) \to (\mathsf{Int} \to \mathsf{Term})$. Figure 4 shows the required changes to the typeful representation of Figure 2.

We can now define `rra` using `coerce` and `uncoerce`. The complete implementation is shown in Figure 5. Types are represented as in Section 5, but the types of the represented types differ. We define the instance as follows.

$$\begin{array}{rcl} [\alpha]_1 & = & \mathsf{Exp}(\alpha) \\ [t_0 \to t_1]_1 & = & [t_0]_1 \to [t_1]_1 \end{array}$$

Then the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_1)$. Haskell infers the following type for the main function.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Exp}(\alpha)$$

```
module TypefulTdpe where

  import TypefulExpCoerce  -- from Figure 4

  data Reify_Reflect(a) =
    RR { reify   :: a -> Exp a,
         reflect :: Exp a -> a }

  rra =                    -- atomic types
    RR { reify   = \x -> coerce x,
         reflect = \x -> uncoerce x }

  rrf (t1, t2) =           -- function types
    RR { reify   = \v -> lam (\x -> reify t2 (v (reflect t1 x))),
         reflect = \e -> \x -> reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 5: A typeful implementation of type-directed partial evaluation

This type proves that type-directed partial evaluation preserves types.

N.B. The typeless implementation in Figure 3 and the typeful implementation in Figure 5 are as efficient. Indeed, they differ only in the two occurrences of `coerce` and `uncoerce` in `rra` in Figure 5, which are defined as the identity function.

6.2 Typeful type-directed partial evaluation (second variant)

The two auxiliary functions `coerce` and `uncoerce` are only necessary to obtain an automatic proof of the type-preservation property of type-directed partial evaluation: They are artefacts of the typeful encoding. But could one do without them? In this section, we present an alternative proof of the typing of type-directed partial evaluation without using these coercions. Instead, we show that when type-directed partial evaluation is applied to a correct representation of the type of the input value, the residual term has the same type as the input value.

To this end, we implement `rra` as a pair of identity functions, as in Figure 3, and we modify the data type Reify_Reflect by weakening the connection between the domains and the codomains of the reify / reflect pairs.

```
module TypefulTdpe where
  import TypefulExp  -- from Figure 2
```

```
data Reify_Reflect a b =
  RR { reify   :: a -> Exp b,
       reflect :: Exp b -> a }

[...]
```

These changes make all of `rra`, `rrf`, and `normalize` well-typed in Haskell. Their types read as follows.

$$
\begin{array}{rcl}
\texttt{rra} & :: & \mathsf{Reify\_Reflect}(\mathsf{Exp}(\alpha))(\alpha) \\
\texttt{rrf} & :: & (\mathsf{Reify\_Reflect}(\alpha)(\gamma), \mathsf{Reify\_Reflect}(\beta)(\delta)) \rightarrow \\
 & & \qquad\qquad \mathsf{Reify\_Reflect}(\alpha \rightarrow \beta)(\gamma \rightarrow \delta) \\
\texttt{normalize} & :: & \mathsf{Reify\_Reflect}(\alpha)(\beta) \rightarrow \alpha \rightarrow \mathsf{Exp}(\beta)
\end{array}
$$

The type of `normalize` no longer proves that it preserves types. However, we can fill in the details by hand using the inferred types of `rra` and `rrf`: We prove by induction on the type t that the type of $|t|$ is $\mathsf{Reify\_Reflect}([t]_1)(t)$. For $t = \alpha$, we have $|t| = \texttt{rra}$ which has type $\mathsf{Reify\_Reflect}(\mathsf{Exp}(\alpha))(\alpha)$ as required. For $t = t_1 \rightarrow t_2$, we have $|t| = \texttt{rrf(}|t_1|\texttt{, }|t_2|\texttt{)}$. By hypothesis, $|t_i|$ has type $\mathsf{Reify\_Reflect}([t_i]_1)(t_i)$ for $i \in \{1, 2\}$. Hence, by the inferred type for `rrf` we have that $\texttt{rrf(}|t_1|\texttt{, }|t_2|\texttt{)}$ has type $\mathsf{Reify\_Reflect}([t_1]_1 \rightarrow [t_2]_1)(t_1 \rightarrow t_2)$ as required. As a corollary we obtain that for all types t,

$$\texttt{normalize } |t| :: [t]_1 \rightarrow \mathsf{Exp}(t)$$

This proof gives a hint about how to prove (by hand) that typeless type-directed partial evaluation preserves types.

6.3 Example: Church numerals, typefully

Let us revisit the example of Section 5.2. We specialize the addition function with respect to a fixed argument using the two typeful variants of type-directed partial evaluation. In both cases the residual terms are the same as in Section 5.2. The Haskell expression `normalize` $|t_{\text{add}}|$ has type $[t_{\text{add}}]_1 \rightarrow \mathsf{Exp}([t_{\text{add}}]_1)$ using the first variant and it has type $[t_{\text{add}}]_1 \rightarrow \mathsf{Exp}(t_{\text{add}})$ using the second variant.

7 Application 2: normal forms

In this section, we use the type inferencer of Haskell as a theorem prover to show that type-directed partial evaluation yields long beta-eta normal forms. We first specify long beta-eta normal forms, both typelessly and typefully (Section 7.1). Then we revisit type-directed partial evaluation, both typelessly (Section 7.2) and typefully (Sections 7.3 and 7.4).

```
module TypelessNf where

  data Nf_ = AT_ At_
           | LAM String Nf_
  data At_ = VAR String
           | APP At_ Nf_

  type Nf = Int -> Nf_
  type At = Int -> At_

  app e1 e2  x = APP (e1 x) (e2 x)
  lam f      x = LAM v (f (\_ -> VAR v) (x + 1))
                 where v = "x" ++ show x
  at2nf e x = AT_ (e x)
```

Figure 6: Typeless representation of normal forms

```
module TypefulNf where

  data Nf_ = AT_ At_
           | LAM String Nf_
  data At_ = VAR String
           | APP At_ Nf_

  data Nf t = NF (Int -> Nf_)
  data At t = AT (Int -> At_)

  app      :: At (a -> b) -> Nf a -> At b
  lam      :: (At a -> Nf b) -> Nf (a -> b)

  coerce   :: Nf a -> Nf (Nf a)
  uncoerce :: At (Nf a) -> Nf a

  at2nf    :: At a -> Nf a

  app (AT e1) (NF e2) = AT (\x -> APP (e1 x) (e2 x))
  lam f               = NF (\x -> let v    = "x" ++ show x
                                      NF b = f (AT (\_ -> VAR v))
                                  in  LAM v (b (x + 1)))

  coerce   (NF f) = NF f
  uncoerce (AT f) = NF (\x -> AT_ (f x))

  at2nf    (AT f) = NF (\x -> AT_ (f x))
```

Figure 7: Typeful representation of normal forms

7.1 Long beta-eta normal forms

We consider explicitly typed λ-terms:

$$\begin{array}{llll} \text{(Types)} & t & ::= & \mathsf{a} \mid t_1 \to t_2 \\ \text{(Terms)} & e & ::= & x \mid e_0\, e_1 \mid \lambda x :: t.\, e \end{array}$$

Definition 1 (long beta-eta normal forms [3, 4]) *A closed term e of type t is in* long beta-eta normal form *if and only if it satisfies $\cdot \vdash_{\mathrm{nf}} e :: t$ where "$\cdot$" denotes the empty environment and where terms in normal form and atomic form are defined by the following rules:*

$$\frac{\Delta, x :: t_1 \vdash_{\mathrm{nf}} e :: t_2}{\Delta \vdash_{\mathrm{nf}} \lambda x :: t_1.\, e :: t_1 \to t_2}[\mathsf{lam}] \qquad \frac{\Delta \vdash_{\mathrm{at}} e :: \mathsf{a}}{\Delta \vdash_{\mathrm{nf}} e :: \mathsf{a}}[\mathsf{coerce}]$$

$$\frac{\Delta \vdash_{\mathrm{at}} e_0 :: t_1 \to t_2 \quad \Delta \vdash_{\mathrm{nf}} e_1 :: t_1}{\Delta \vdash_{\mathrm{at}} e_0\, e_1 :: t_2}[\mathsf{app}] \qquad \frac{\Delta(x) = t}{\Delta \vdash_{\mathrm{at}} x :: t}[\mathsf{var}]$$

No term containing β-redexes can be derived by these rules, and the coerce rule ensures that the derived terms are fully η-expanded.

Figure 6 displays a typeless representation of normal forms in Haskell. Figure 7 displays a typeful representation of normal forms in Haskell.

7.2 Typeless type-directed partial evaluation and normal forms

We now reexpress type-directed partial evaluation as specified in Figure 8 to yield typeless terms, as also done by Filinski [3]. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Nf}$$

This type proves that type-directed partial evaluation yields residual terms in beta normal form since the representation of Figure 6 does not allow beta redexes. These residual terms are also in eta normal form because `at2nf` is only applied at base type: residual terms are thus fully eta expanded.

7.3 Typeful type-directed partial evaluation and normal forms (first variant)

We now reexpress type-directed partial evaluation to yield typeful terms as specified in Figure 9. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha) \to \alpha \to \mathsf{Nf}(\alpha)$$

This type proves that type-directed partial evaluation (1) preserves types and (2) yields terms in normal form.

```
module TypelessTdpeNf where

  import TypelessNf  -- from Figure 6

  data Reify_Reflect a =
    RR { reify   :: a -> Nf,
         reflect :: At -> a }

  rra =              -- atomic types
    RR { reify   = \x -> x,
         reflect = \x -> at2nf x }

  rrf (t1, t2) =     -- function types
    RR { reify   = \v -> lam (\x -> reify t2 (v (reflect t1 x))),
         reflect = \e -> \x -> reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 8: Typeless implementation of type-directed partial evaluation with normal forms

```
module TypefulTdpeNf1 where

  import TypefulNf  -- from Figure 7

  data Reify_Reflect a =
    RR { reify   :: a -> Nf a,
         reflect :: At a -> a }

  rra =             -- atomic types
    RR { reify   = \x -> coerce x,
         reflect = \x -> uncoerce x }

  rrf (t1, t2) =    -- function types
    RR { reify   = \v -> lam (\x -> reify t2 (v (reflect t1 x))),
         reflect = \e -> \x -> reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 9: Typeful implementation of type-directed partial evaluation with normal forms (first variant)

```
module TypefulTdpeNf2 where

  import TypefulNf  -- from Figure 7

  data Reify_Reflect a b =
    RR { reify   :: a -> Nf b,
         reflect :: At b -> a }

  rra =             -- atomic types
    RR { reify   = \x -> x,
         reflect = \x -> at2nf x }

  rrf (t1, t2) =    -- function types
    RR { reify   = \v -> lam (\x -> reify t2 (v (reflect t1 x))),
         reflect = \e -> \x -> reflect t2 (app e (reify t1 x)) }

  normalize t v = reify t v
```

Figure 10: Typeful implementation of type-directed partial evaluation with normal forms (second variant)

7.4 Typeful type-directed partial evaluation and normal forms (second variant)

On the same ground as Section 6.2, i.e., to bypass the artefactual coercions of the typeful encoding of abstract syntax, we now reexpress type-directed partial evaluation to yield typeful terms as specified in Figure 10. The type of `normalize` reads as follows.

$$\texttt{normalize} :: \mathsf{Reify\_Reflect}(\alpha)(\beta) \rightarrow \alpha \rightarrow \mathsf{Nf}(\beta)$$

This type only proves that type-directed partial evaluation yields terms in normal form. As in Section 6.2, we can prove type preservation by hand, i.e., that

$$\texttt{normalize}\ |t| :: [t]_2 \rightarrow \mathsf{Nf}(t)$$

where the instance of a type is defined by

$$\begin{array}{rcl} [\alpha]_2 & = & \mathsf{Nf}(\alpha) \\ [t_1 \rightarrow t_2]_2 & = & [t_1]_2 \rightarrow [t_2]_2 \end{array}$$

8 Conclusions and issues

We have presented a simple way to express typed abstract syntax in a Haskell-like language, and we have used this typed abstract syntax to demonstrate that type-directed partial evaluation preserves types and yields residual programs in normal form. The encoding is limited because it does not lend itself to programs taking typed abstract syntax as input—as, e.g., a typeful transformation into continuation-passing style. Nevertheless, the encoding is sufficient to establish two key properties of type-directed partial evaluation automatically.

These two properties could be illustrated more directly in a language with dependent types such as Martin-Löf type theory. In such a language, one can directly represent typed abstract syntax and program type-directed partial evaluation typefully.

Acknowledgments: This work is supported by the ESPRIT Working Group APPSEM (`http://www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`). Part of it was carried out while the second author was visiting Jason Hickey at Caltech, in the summer and fall of 2000. We are grateful to the anonymous reviewers and to Julia Lawall for perceptive comments.

A ML programs

In this appendix we present the ML implementation of the programs in the body of this report. The main differences between the ML programs and the Haskell programs are as follows.

- The ML programs use `gensym` to generate fresh variable names instead of a threaded variable. (Compare the figures 1A, 2A, 6A, and 7A with their corresponding Haskell counterparts.) It is possible to use a threaded variable to generate fresh variable names in the ML programs, too. On the other hand, Haskell is a pure functional language, so it is not possible to use `gensym` to generate fresh variable names in the Haskell programs. The implementation of `gensym` is standard:

```
structure Gensym
= struct
   val gensym_count = ref 0

   fun gensym ()
       = let val this = !gensym_count
         in  gensym_count := this + 1;
             "x" ^ (Int.toString this)
         end
   end
```

- The ML programs use signatures to restrict the types of the higher-order constructors instead of using a datatype. (Compare the figures 2A, 4A, and 7A with their corresponding Haskell counterparts.) It is possible to use a datatype to restrict the types of the higher-order constructors in the ML programs, too. On the other hand, Haskell does not have signatures, so this modular approach cannot be used in the Haskell programs.

```
structure TypelessExp
= struct
    datatype exp = INT of int
                 | VAR of string
                 | APP of exp * exp
                 | LAM of string * exp

    val int = INT
    val app = APP
    fun lam f
        = let val x = Gensym.gensym ()
          in  LAM(x, f(VAR x))
          end
  end
```

Figure 1A: Typeless higher-order abstract syntax in ML

```
structure TypefulExp
:> sig
     type 'a exp

     val int : int -> int exp
     val app : ('a -> 'b) exp * 'a exp -> 'b exp
     val lam : ('a exp -> 'b exp) -> ('a -> 'b) exp
   end
= struct
    datatype term = INT of int
                  | VAR of string
                  | APP of term * term
                  | LAM of string * term

    type 'a exp = term

    val int = INT
    val app = APP
    fun lam f
        = let val x = Gensym.gensym ()
          in  LAM(x, f(VAR x))
          end
  end
```

Figure 2A: Typeful higher-order abstract syntax in ML

```
structure TypelessTdpe
= let open TypelessExp           (* from Fig. 1A *)
  in struct
       datatype 'a rr = RR of { reify   : 'a -> exp,
                                reflect : exp -> 'a }

       val rra                   (* atomic types *)
           = RR { reify   = fn x => x,
                  reflect = fn x => x }

       fun rrf (RR t1, RR t2)    (* function types *)
           = RR { reify
                  = fn v => lam (fn x => #reify t2 (v (#reflect t1 x))),
                  reflect
                  = fn e => fn x => #reflect t2 (app(e, #reify t1 x)) }

       fun normalize (RR t) v = #reify t v
     end
  end
```

Figure 3A: A typeless implementation of type-directed partial evaluation

```
structure TypefulExpCoerce
:> sig
     [...]
     val coerce   : 'a exp -> ('a exp) exp
     val uncoerce : ('a exp) exp -> 'a exp
   end
= struct
    [...]
    fun coerce   e = e
    fun uncoerce e = e
  end
```

Figure 4A: Typeful higher-order abstract syntax
with coercions for atomic types

```
structure TypefulTdpeCoerce
= let open TypefulExpCoerce      (* from Fig. 4A *)
  in struct
       datatype 'a rr = RR of { reify   : 'a -> 'a exp,
                                reflect : 'a exp -> 'a }

       val rra                   (* atomic types *)
           = RR { reify   = fn x => coerce x,
                  reflect = fn x => uncoerce x }

       fun rrf (RR t1, RR t2)    (* function types *)
           = RR { reify
                  = fn v => lam (fn x => #reify t2 (v (#reflect t1 x))),
                  reflect
                  = fn e => fn x => #reflect t2 (app(e, #reify t1 x)) }

       fun normalize (RR t) v = #reify t v
     end
  end
```

Figure 5A: A typeful implementation of type-directed partial evaluation

```
structure TypelessNf
= struct
    datatype nf = AT of at
                | LAM of string * nf
         and at = VAR of string
                | APP of at * nf

    val app = APP
    fun lam f
        = let val x = Gensym.gensym ()
          in  LAM(x, f(VAR x))
          end

    val at2nf = AT
  end
```

Figure 6A: Typeless representation of normal forms

```
structure TypefulNf
:> sig
     type 'a nf
     type 'a at

     val app : ('a -> 'b) at * 'a nf -> 'b at
     val lam : ('a at -> 'b nf) -> ('a -> 'b) nf
     val coerce   : 'a nf -> ('a nf) nf
     val uncoerce : ('a nf) at -> 'a nf
     val at2nf    : 'a at -> 'a nf
   end
= struct
    datatype nf_ = AT  of at_
                 | LAM of string *  nf_
         and at_ = VAR of string
                 | APP of at_ * nf_

    type 'a nf = nf_
    type 'a at = at_

    val app = APP
    fun lam f
        = let val v = Gensym.gensym ()
          in  LAM(v, f (VAR v))
          end

    fun coerce   f = f
    fun uncoerce f = AT f

    val at2nf = AT
  end
```

Figure 7A: Typeful representation of normal forms

```
structure TypelessTdpeNf
= let open TypelessNf           (* from Fig. 6A *)
  in struct
       datatype 'a rr = RR of { reify   : 'a -> nf,
                                reflect : at -> 'a }

       val rra                  (* atomic types *)
           = RR { reify   = fn x => x,
                  reflect = fn x => at2nf x }

       fun rrf (RR t1, RR t2)   (* function types *)
           = RR { reify
                  = fn v => lam (fn x => #reify t2 (v (#reflect t1 x))),
                  reflect
                  = fn e => fn x => #reflect t2 (app(e, #reify t1 x)) }

       fun normalize (RR t) v = #reify t v
  end
end
```

Figure 8A: Typeless implementation of type-directed partial evaluation with normal forms

```
structure TypefulTdpeNfCoerce
= let open TypefulNf            (* from Fig. 7A *)
  in struct
       datatype 'a rr = RR of { reify   : 'a -> 'a nf,
                                reflect : 'a at -> 'a }

       val rra                  (* atomic types *)
           = RR { reify   = fn x => coerce x,
                  reflect = fn x => uncoerce x }

       fun rrf (RR t1, RR t2)   (* function types *)
           = RR { reify
                  = fn v => lam (fn x => #reify t2 (v (#reflect t1 x))),
                  reflect
                  = fn e => fn x => #reflect t2 (app(e, #reify t1 x)) }

       fun normalize (RR t) v = #reify t v
     end
  end
```

Figure 9A: Typeful implementation of type-directed partial evaluation with normal forms (first variant)

```
structure TypefulTdpeNfWeak
= let open TypefulNf          (* from Fig. 7A *)
  in struct
       datatype ('a, 'b) rr = RR of { reify   : 'a -> 'b nf,
                                      reflect : 'b at -> 'a }

       val rra                (* atomic types *)
           = RR { reify   = fn x => x,
                  reflect = fn x => at2nf x }

       fun rrf (RR t1, RR t2) (* function types *)
           = RR { reify
                  = fn v => lam (fn x => #reify t2 (v (#reflect t1 x))),
                  reflect
                  = fn e => fn x => #reflect t2 (app(e, #reify t1 x)) }

       fun normalize (RR t) v = #reify t v
     end
  end
```

Figure 10A: Typeful implementation of type-directed partial evaluation with normal forms (second variant)

References

[1] Olivier Danvy. Type-directed partial evaluation. In John Hatcliff, Torben Æ. Mogensen, and Peter Thiemann, editors, *Partial Evaluation – Practice and Theory; Proceedings of the 1998 DIKU Summer School*, number 1706 in Lecture Notes in Computer Science, pages 367–411, Copenhagen, Denmark, July 1998. Springer-Verlag.

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

[3] 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. Extended version available as the technical report BRICS RS-99-17.

[4] Gérard Huet. Résolution d'équations dans les langages d'ordre 1, 2, ..., ω. Thèse d'État, Université de Paris VII, Paris, France, 1976.

[5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`.

[6] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.

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

[8] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 23, No 7, pages 199–208, Atlanta, Georgia, June 1988. ACM Press.

[9] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical report BRICS-NS-99-1, University of Aarhus, pages 25–29, San Antonio, Texas, January 1999.

[10] 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 the technical report BRICS RS-98-9.

Recent BRICS Report Series Publications