JFP **11** (6): 673–680, November 2001. © 2001 Cambridge University Press 673
DOI: 10.1017/S0956796801004166 Printed in the United Kingdom

FUNCTIONAL PEARLS

Normalization by evaluation with typed abstract syntax

OLIVIER DANVY, MORTEN RHIGER
BRICS, Department of Computer Science, University of Aarhus, Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark*
(*e-mail:* {danvy,mrhiger}@brics.dk)
(*Home pages:* http://www.brics.dk/~{danvy,mrhiger})

KRISTOFFER H. ROSE
IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, USA
(*e-mail:* krisrose@us.ibm.com)

1 A write-only typed abstract syntax

In higher-order abstract syntax, the variables and bindings of an object language are represented by variables and bindings of a meta-language. Let us consider the simply typed λ-calculus as object language and Haskell as meta-language. For concreteness, we also throw in integers and addition, but only in this section.

```
data Term = INT Int | ADD Term Term
          | APP Term Term | LAM (Term → Term)
```

The constructors are typed as follows.

```
INT :: Int  →  Term
ADD :: Term  →  Term  →  Term
APP :: Term  →  (Term → Term)
LAM :: (Term → Term)  →  Term
```

They do not prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `LAM(λx→APP x x)` yields a value of type `Term`.

We can, however, provide a typed interface to these constructors preventing us from forming ill-typed terms.

```
newtype Exp t = EXP Term

int :: Int  →  Exp Int
int i = EXP (INT i)
```

* Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

```
add :: Exp Int → Exp Int → Exp Int
add (EXP e1) (EXP e2) = EXP (ADD e1 e2)

app :: Exp (a→b) → (Exp a → Exp b)
app (EXP e1) (EXP e2) = EXP (APP e1 e2)

lam :: (Exp a → Exp b) → Exp (a→b)
lam f = EXP (LAM (λx→let EXP b = f (EXP x) in b))
```

The type `Exp` is parameterized over a type `t` but does not use it: `t` is a *phantom type*.

These typeful constructors prevent us from forming ill-typed terms. For example, in the scope of these constructors, evaluating `lam(λx→app x x)` yields a type error. Conversely, if a term has the simple type `t` then its typed abstract-syntax representation has type `Exp t`, which can be illustrated as follows:

```
λx→x + 5                  :: Int→Int
lam (λx→add x (int 5))    :: Exp (Int→Int)
```

We intend to use this typed abstract syntax to show that normalization by evaluation preserves types (Section 2) and yields normal forms (Section 3) for the pure and simply typed λ-calculus. Therefore, we are only interested in constructing abstract syntax. (To convert a constructed term into first-order abstract syntax where variables are represented as strings, one needs to add another constructor to `Term` for free variables.) Furthermore, such a write-only typed abstract syntax does not solve the basic problem of programming higher-order abstract syntax in Haskell, which is that the function space in the `LAM` summand is 'too big', in the sense that it allows both non-strict and non-total functions. But again, this representation is sufficient for our purpose here. In the remainder of this pearl, `Term` and `Exp` are restricted to the pure λ-calculus.

2 Normalization by evaluation preserves types

Normalization by evaluation is an extensional, reduction-free technique for strongly normalizing closed λ-terms. Source terms are represented as meta-language values and a *normalization function* maps these values into a syntactic representation of their normal form.

The technique is extensional instead of intensional because the source terms are (higher-order) values, not (first-order) symbolic representations. It is reduction-free because all the β-reductions needed to yield a normal form are carried out implicitly by the underlying implementation of the meta-language. For this reason, it runs at native speed, and thus is more efficient than traditional, symbolic normalization.

Normalization by evaluation uses two type-indexed and mutually recursive functions. One, *reify*, traditionally noted $\downarrow$, maps a value into its representation and the other, *reflect*, traditionally noted $\uparrow$, maps a representation into a value. These two functions are canonically defined as follows, for the simply typed λ-calculus.

https://doi.org/10.1017/S0956796801004166 Published online by Cambridge University Press

$$\begin{array}{rcl} t & ::= & \alpha \mid t_1 \to t_2 \\ \downarrow^{\alpha} & = & \overline{\lambda} v.v \\ \downarrow^{t_1 \to t_2} & = & \overline{\lambda} v.\underline{\lambda} x.\downarrow^{t_2} \overline{@} (v \overline{@} (\uparrow_{t_1} \overline{@} x)) \\ \uparrow_{\alpha} & = & \overline{\lambda} e.e \\ \uparrow_{t_1 \to t_2} & = & \overline{\lambda} e.\overline{\lambda} x.\uparrow_{t_2} \overline{@} (e \underline{@} (\downarrow^{t_1} \overline{@} x)) \end{array}$$

where overlined λ and @ denote meta-level abstractions and applications, respectively, and underlined λ and @ denote object-level abstractions and applications.

A simply typed term is normalized by reifying its value. For example, let us consider Church numbers.

$$\begin{array}{rcl} zero & = & \overline{\lambda} s.\overline{\lambda} z.z \\ succ & = & \overline{\lambda} n.\overline{\lambda} s.\overline{\lambda} z.s \overline{@} (n \overline{@} s \overline{@} z) \\ three & = & succ \overline{@} (succ \overline{@} (succ \overline{@} zero)) \\ add & = & \overline{\lambda} m.\overline{\lambda} n.\overline{\lambda} s.\overline{\lambda} z.m \overline{@} s \overline{@} (n \overline{@} s \overline{@} z) \end{array}$$

Reifying *three* yields $\underline{\lambda} s.\underline{\lambda} z.s \underline{@} (s \underline{@} (s \underline{@} z))$, i.e., the representation in normal form of 3. Similarly, reifying $add \overline{@} zero$ yields $\underline{\lambda} n.\underline{\lambda} s.\underline{\lambda} z.n \underline{@} (\underline{\lambda} n'.s \underline{@} n') \underline{@} z$, i.e., the representation in long $\beta\eta$-normal form of the identity function over Church numbers, reflecting that zero is neutral for addition. And finally, reifying $add \overline{@} three$ yields the representation in normal form of a function iterating the successor function three times, i.e., $\underline{\lambda} n.\underline{\lambda} s.\underline{\lambda} z.s \underline{@} (s \underline{@} (s \underline{@} (n \underline{@} (\underline{\lambda} n'.s \underline{@} n') \underline{@} z)))$. The source terms are values (i.e., with overlined λ and @) and, using $\downarrow$, we have reified them into a syntactic representation of their normal form (i.e., with underlined λ and @).

The type of a Church number is `(a→a) → a → a`. The type of its normal form is `Term`, or, perhaps more vividly, `Exp ((a → a) → a → a)`.

Normalization by evaluation is defined by induction on the structure of types, which makes it a natural candidate to be expressed with type classes. We thus define a type class `Nbe` hosting two type-indexed functions, `reify` and `reflect`. Representing object terms with the type `Term` of Section 1 would give us the usual uninformative type `t→Term` for `reify` and `Term→t` for `reflect`. Instead, let us use the parameterized type `Exp` of Section 1:

```
class Nbe a
  where  reify   :: a → Exp a
         reflect :: Exp a → a
```

The challenge now is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types.

The canonical definition above dictates how to instantiate `Nbe` at function type.

```
instance  (Nbe a, Nbe b) ⇒ Nbe (a→b)
  where  reify   v = lam (λx→reify (v (reflect x)))
         reflect e = λx→reflect (app e (reify x))
```

https://doi.org/10.1017/S0956796801004166 Published online by Cambridge University Press

For base types, reify and reflect are two identity functions. To be type correct, however, reify must produce a term and reflect must consume a term. We can ensure that reify produces a term when its argument is a term. Similarly, we can ensure that reflect consumes a term when its result is a term. Taking advantage of the fact that the type parameter of Exp is a phantom type, we thus introduce the following two 'phantom' identity functions for the base case:

```
coerce :: Exp (Exp a) → Exp a
coerce (EXP v) = EXP v

uncoerce :: Exp a → Exp (Exp a)
uncoerce (EXP e) = EXP e

instance Nbe (Exp a)
  where reify = uncoerce
        reflect = coerce
```

A value v is normalized by applying reify to it. In usual implementations of normalization by evaluation, (a representation of) the type of v must be supplied on par with v, as an input data. Here, because we use type classes, this type is supplied as a cast, to resolve overloading. It is obtained by instantiating type variables a with Exp a, in the original type. So for example, id . id has the type a→a. Reifying it at type Exp a → Exp a yields λx→x, and reifying it at type (Exp a → Exp a) → (Exp a → Exp a) yields λx→λx'→x x'.

3 Normalization by evaluation yields normal forms

In the simply typed λ-calculus, long $\beta\eta$-normal forms are closed terms without β-redexes that are fully η-expanded with respect to their type. A closed term e of type t and in normal form satisfies $\vdash_{\mathrm{nf}} e :: t$, where terms in normal form (and atomic form) are defined by the following rules:

$$\frac{\Delta, x :: t_1 \vdash_{\mathrm{nf}} e :: t_2}{\Delta \vdash_{\mathrm{nf}} (\lambda x :: t_1 . e) :: t_1 \rightarrow t_2}(\mathrm{Lam}) \qquad \frac{\Delta \vdash_{\mathrm{at}} e :: \alpha}{\Delta \vdash_{\mathrm{nf}} e :: \alpha}(\mathrm{Coerce})$$

$$\frac{\Delta \vdash_{\mathrm{at}} e_0 :: t_1 \rightarrow t_2 \quad \Delta \vdash_{\mathrm{nf}} e_1 :: t_1}{\Delta \vdash_{\mathrm{at}} e_0\, e_1 :: t_2}(\mathrm{App}) \qquad \frac{\Delta(x) = t}{\Delta \vdash_{\mathrm{at}} x :: t}(\mathrm{Var})$$

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

As in Section 1, we provide a typed interface to the constructors of terms in normal form, preventing us from forming ill-typed terms.

```
data NfTerm = COERCE AtTerm | LAM (AtTerm → NfTerm)
data AtTerm = APP AtTerm NfTerm

newtype NfExp a = NF NfTerm
newtype AtExp a = AT AtTerm
```

https://doi.org/10.1017/S0956796801004166 Published online by Cambridge University Press

```
app' :: AtExp (a→b) → (NfExp a → AtExp b)
app' (AT e1) (NF e2) = AT (APP e1 e2)

lam' :: (AtExp a → NfExp b) → NfExp (a→b)
lam' f = NF (LAM (λx→let NF t = f (AT x) in t))

coerce' :: AtExp (NfExp a) → NfExp a
coerce' (AT v) = NF (COERCE v)

uncoerce' :: NfExp a → NfExp (NfExp a)
uncoerce' (NF e) = NF e
```

These declarations specialize the representation from Section 2 to reflect that the represented terms are in normal form. As in Section 2, we provide two phantom identity functions, `coerce'` and `uncoerce'`, where `coerce'` constructs terms that arise from using the above Coerce rule.

Thus equipped, we can re-express normalization by evaluation in an implementation that yields a representation of λ-terms in normal form.

```
class  Nbe' a
  where  reify :: a → NfExp a
         reflect :: AtExp a → a
```

Again, the challenge is to populate this type class with values of function type and of base type implementing normalization by evaluation. If we can do that, the type inferencer of Haskell will act as a theorem prover and will demonstrate that this implementation of normalization by evaluation preserves types and yields normal forms.

The instances use the constructors for terms in normal forms but are otherwise defined as in Section 2.

```
instance  (Nbe' a, Nbe' b) ⇒ Nbe' (a→b)
  where  reify   v = lam' (λx→reify (v (reflect x)))
         reflect e = λx→reflect (app' e (reify x))

instance  Nbe' (NfExp a)
  where  reify = uncoerce'
         reflect = coerce'
```

As in Section 2, reifying `id . id` at type `NfExp a → NfExp a` yields `λx→x`, and reifying it at type `(NfExp a → NfExp a) → (NfExp a → NfExp a)` yields `λx→λx'→x x'`.

For a last example, here are the Haskell definitions of Church numbers mentioned in Section 2.

```
type Number a = (a → a) → a → a
zero  = λs z→z
succ  = λn s z→s (n s z)
three = succ (succ (succ zero))
add   = λm n s z→m s (n s z)
```

Reifying `three`, `add zero`, and `add three` gives the text of their normal form at type `Number (Exp a) → Number (Exp a)`.

https://doi.org/10.1017/S0956796801004166 Published online by Cambridge University Press

4 Conclusions and issues

We have presented a simple encoding of typed abstract syntax in Haskell, and we have used this typed abstract syntax to demonstrate that normalization by evaluation preserves simple types and yields residual programs in $\beta\eta$-normal form. The encoding is write-only because it does not lend itself to programs taking typed abstract syntax as input, such as a typed transformation into continuation-passing style. Nevertheless, it is sufficient to establish two key properties of normalization by evaluation automatically, using the Haskell type inferencer as a theorem prover.

These two properties could be illustrated more directly in a language with dependent types such as Martin-Löf's type theory. In such a language, one can directly embed simply typed λ-terms (in normal form or not), express normalization by evaluation, and prove that it preserves types and yields normal forms.

Related work Normalization by evaluation takes its roots in type theory (Coquand & Dybjer, 1997; Martin-Löf, 1975), proof theory (Berger, 1993; Berger *et al*., 1998; Berger & Schwichtenberg, 1991), logic (Altenkirch *et al*., 1996), category theory (Altenkirch *et al*., 1995; Čubrić *et al*., 1998; Reynolds, 1998), and partial evaluation (Danvy, 1998; Filinski, 1999; Rhiger, 1999; Rose, 1998). Long $\beta\eta$-normal forms were specified, for example, in Huet's thesis (Huet, 1976). The particular characterization we use originates in Pfenning's work on Logical Frameworks, and so does higher-order abstract syntax (Pfenning & Elliott, 1988). We use it further to pair normalization by evaluation and run-time code generation (Balat & Danvy, 1998; Rhiger, 2001). Our typed abstract syntax is akin to Leijen and Meijer's embedding of SQL into Haskell, which introduced phantom types (Leijen & Meijer, 1999). Phantom types provide a typing discipline for otherwise untyped values such as pointers in a foreign language interface (Finne *et al*., 1999).

Acknowledgements

A preliminary and longer version of this article is available in the proceedings of *FLOPS 2001* (Danvy & Rhiger, 2001). We would like to thank Simon Peyton Jones for identifying phantom types in it. The present version has benefited from Richard Bird's editorial advice and from Ralf Hinze's comments.

Part of this work was carried out while the second author was visiting Jason Hickey at Caltech, in the summer and fall of 2000, and while the third author was affiliated with BRICS, in 1996-1997. We are supported by the ESPRIT Working Group APPSEM (`www.md.chalmers.se/Cs/Research/Semantics/APPSEM/`).

References

Altenkirch, T., Hofmann, M. & Streicher, T. (1995) Categorical reconstruction of a reduction-free normalization proof. In: Pitt, D. H., Rydeheard, D. E. & Johnstone, P. (eds.), *Category Theory and Computer Science: Lecture Notes in Computer Science 953*, pp. 182–199. Springer-Verlag.

Altenkirch, T., Hofmann, M. & Streicher, T. (1996) Reduction-free normalisation for a polymorphic system. In: Clarke E. M. (ed.), *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pp. 98–106. IEEE Press.

Balat, V. & Danvy, O. (1998) Strong normalization by type-directed partial evaluation and run-time code generation. In: Leroy, X. & Ohori, A. (eds.), *Proceedings 2nd International Workshop on Types in Compilation: Lecture Notes in Computer Science 1473*, pp. 240–252. Springer-Verlag.

Berger, U. (1993) Program extraction from normalization proofs. In: Bezem, M. & Groote, J. F. (eds.), *Typed Lambda Calculi and Applications: Lecture Notes in Computer Science 664*, pp. 91–106. Springer-Verlag.

Berger, U. & Schwichtenberg, H. (1991) An inverse of the evaluation functional for typed λ-calculus. In: Kahn, G. (ed), *Proceedings 6th Annual IEEE Symposium on Logic in Computer Science*, pp. 203–211. IEEE Press.

Berger, U., Eberl, M. & Schwichtenberg, H. (1998) Normalization by evaluation. In: Möller, B. & Tucker, J. V. (eds.), *Prospects for Hardware Foundations (NADA): Lecture Notes in Computer Science 1546*, pp. 117–137. Springer-Verlag.

Coquand, T. & Dybjer, P. (1997) Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, **7**, 75–94.

Čubrić, D., Dybjer, P. & Scott, P. (1998) Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, **8**, 153–192.

Danvy, O. (1998) Type-directed partial evaluation. In: Hatcliff, J., Mogensen, T. Æ. & Thiemann, P. (eds.), *Partial Evaluation – Practice and Theory; proceedings of the 1998 DIKU Summer School: Lecture Notes in Computer Science 1706*, pp. 367–411. Springer-Verlag.

Danvy, O. & Dybjer, P. (eds.) (1998) *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, Chalmers, Sweden. BRICS Note Series, nos. NS–98–1.

Danvy, O. & Rhiger, M. (2001) A simple take on typed abstract syntax in Haskell-like languages. In: Kuchen, H. & Ueda, K. (eds)., *5th International Symposium on Functional and Logic Programming: Lecture Notes in Computer Science 2024*, pp. 343–358. Springer-Verlag.

Filinski, A. (1999) A semantic account of type-directed partial evaluation. In: Nadathur, G. (ed.), *Proceedings of the International Conference on Principles and Practice of Declarative Programming: Lecture Notes in Computer Science 1702*, pp. 378–395. Springer-Verlag.

Finne, S., Leijen, D., Meijer, E. & Peyton Jones, S. (1999) Calling hell from heaven and heaven from hell. In: Lee, P. (ed.), *Proceedings 1999 ACM SIGPLAN International Conference on Functional Programming*, pp. 114–125. ACM Press.

Huet, G. (1976) *Résolution d'équations dans les langages d'ordre 1, 2, ..., ω*. Thèse d'État, Université de Paris VII, Paris, France.

Leijen, D. & Meijer, E. (1999) Domain specific embedded compilers. In: Ball, T. (ed.), *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pp. 109–122.

Martin-Löf, P. (1975) About models for intuitionistic type theories and the notion of definitional equality. *Proceedings 3rd Scandinavian Logic Symposium: Studies in Logic and the Foundation of Mathematics 82*, pp. 81–109. North-Holland.

Pfenning, F. & Elliott, C. (1988) Higher-order abstract syntax. In: Schwartz, M. D. (ed.), *Proceedings ACM SIGPLAN'88 Conference on Programming Languages Design and Implementation*, pp. 199–208. (*SIGPLAN Notices*, **23**(7).) ACM Press.

Reynolds, J. C. (1998) Normalization and functor categories. In: Danvy, O. & Dybjer, P. (eds.), *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, Chalmers, Sweden.

Rhiger, M. (1999) Deriving a statically typed type-directed partial evaluator. In: Danvy, O. (ed.), *Proceedings ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 25–29. BRICS Note Series, no. NS–99–1.

Rhiger, M. (2001) PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark. Forthcoming.

Rose, K. (1998) Type-directed partial evaluation using type classes. In: Danvy, O. & Dybjer, P. (eds.), *Preliminary Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation, NBE '98*, Chalmers, Sweden.