BRICS RS-01-16 Danvy et al.: Normalization by Evaluation with Typed Abstract Syntax

BRICS

Basic Research in Computer Science

Normalization by Evaluation with Typed Abstract Syntax

Olivier Danvy
Morten Rhiger
Kristoffer H. Rose

BRICS Report Series RS-01-16

ISSN 0909-0878 May 2001

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

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

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

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/01/16/

Normalization by Evaluation with Typed Abstract Syntax *

Olivier Danvy and Morten Rhiger

BRICS †

Department of Computer Science, University of Aarhus ‡

Kristoffer H. Rose

IBM T. J. Watson Research Center §

May 10, 2001

Abstract

We present a simple way to implement typed abstract syntax for the lambda calculus in Haskell, using phantom types, and we specify normalization by evaluation (i.e., type-directed partial evaluation) to yield this typed abstract syntax. Proving that normalization by evaluation preserves types and yields normal forms then reduces to type-checking the specification.

Contents

*To appear in the Journal of Functional Programming. (Extended version.)

†Basic Research in Computer Science (`www.brics.dk`), funded by 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}`

§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)

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 a 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.

Normalization by evaluation is extensional and reduction-free. It 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.

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

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.

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

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 $\mathit{add} \overline{@} \mathit{zero}$ 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. Finally, reifying $\mathit{add} \overline{@} \mathit{three}$

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))
```

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 $\vdash_{\text{nf}} e :: t$, where terms in normal form (and atomic form) are defined by the following rules.

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

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

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

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 earlier, 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)`.

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—as, e.g., 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 [7, 16], proof theory [4, 5, 6], logic [2], category theory [1, 8, 18], and partial evaluation [9, 12, 19, 21]. Long $\beta\eta$-normal forms were specified, e.g., in Huet's thesis [14]. The particular characterization we use originates in Pfenning's work on Logical Frameworks, and so does higher-order abstract syntax [17]. We use it further to pair normalization by evaluation and run-time code generation [3, 20]. Our typed abstract syntax is akin to Leijen and Meijer's embedding of SQL into Haskell, which introduced phantom types [15]. Phantom types provide a typing discipline for otherwise untyped values such as pointers in a foreign language interface [13].

Acknowledgments: A preliminary and longer version of this article is available in the proceedings of FLOPS 2001 [11]. 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

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

[2] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *Proceedings of the Eleventh Annual IEEE Symposium on Logic in Computer Science*, New Brunswick, New Jersey, July 1996. IEEE Computer Society Press.

[3] Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

[4] Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

[5] Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects*

for hardware foundations (NADA), number 1546 in Lecture Notes in Computer Science, pages 117–137. Springer-Verlag, 1998.

[6] 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.

[7] Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

[8] Djordje Čubrić, Peter Dybjer, and Philip Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

[9] 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.

[10] 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.

[11] Olivier Danvy and Morten Rhiger. A simple take on typed abstract syntax in Haskell-like languages. In Herbert Kuchen and Kazunori Ueda, editors, *Fifth International Symposium on Functional and Logic Programming*, number 2024 in Lecture Notes in Computer Science, pages 343–358, Tokyo, Japan, March 2001. Springer-Verlag. Extended version available as the technical report BRICS RS-00-34.

[12] Andrzej Filinski. A semantic account of type-directed partial evaluation. In Gopalan Nadathur, editor, *Proceedings of the 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.

[13] Sigbjorn Finne, Daan Leijen, Erik Meijer, and Simon Peyton Jones. Calling hell from heaven and heaven from hell. In Peter Lee, editor, *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 114–125, Paris, France, September 1999. ACM Press.

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

[15] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In Thomas Ball, editor, *Proceedings of the 2nd USENIX Conference on Domain-Specific Languages*, pages 109–122, Austin, Texas, October 1999.

[16] Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

[17] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In Mayer D. Schwartz, editor, *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.

[18] John C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [10].

[19] Morten Rhiger. Deriving a statically typed type-directed partial evaluator. In Olivier Danvy, editor, *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.

[20] Morten Rhiger. PhD thesis, BRICS PhD School, University of Aarhus, Aarhus, Denmark, 2001. Forthcoming.

[21] Kristoffer Rose. Type-directed partial evaluation using type classes. In Danvy and Dybjer [10].

Recent BRICS Report Series Publications

RS-01-16 **Olivier Danvy, Morten Rhiger, and Kristoffer H. Rose.** ***Normalization by Evaluation with Typed Abstract Syntax*****. May 2001. 9 pp. To appear in** ***Journal of Functional Programming*****.**

RS-01-15 **Luigi Santocanale.** ***A Calculus of Circular Proofs and its Categorical Semantics*****. May 2001. 30 pp.**

RS-01-14 **Ulrich Kohlenbach and Paulo B. Oliva.** ***Effective Bounds on Strong Unicity in L_1-Approximation*****. May 2001.**

RS-01-13 **Federico Crazzolara and Glynn Winskel.** ***Events in Security Protocols*****. April 2001.**

RS-01-12 **Torben Amtoft, Charles Consel, Olivier Danvy, and Karoline Malmkjær.** ***The Abstraction and Instantiation of String-Matching Programs*****. April 2001.**

RS-01-11 **Alexandre David and M. Oliver Möller.** ***From Hierarichcal Timed Automata to*** **UPPAAL. March 2001.**

RS-01-10 **Daniel Fridlender and Mia Indrika.** ***Do we Need Dependent Types?*** **March 2001. 6 pp. Appears in** ***Journal of Functional Programming*****, 10(4):409–415, 2000. Superseeds BRICS Report RS-98-38.**

RS-01-9 **Claus Brabrand, Anders Møller, and Michael I. Schwartzbach.** ***Static Validation of Dynamically Generated HTML*****. February 2001. 18 pp.**

RS-01-8 **Ulrik Frendrup and Jesper Nyholm Jensen.** ***Checking for Open Bisimilarity in the π-Calculus*****. February 2001. 61 pp.**

RS-01-7 **Gregory Gutin, Khee Meng Koh, Eng Guan Tay, and Anders Yeo.** ***On the Number of Quasi-Kernels in Digraphs*****. January 2001. 11 pp.**

RS-01-6 **Gregory Gutin, Anders Yeo, and Alexey Zverovich.** ***Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*****. January 2001. 7 pp.**

RS-01-5 **Thomas S. Hune, Judi Romijn, Mariëlle Stoelinga, and Frits W. Vaandrager.** ***Linear Parametric Model Checking of Timed Automata*****. January 2001. 44 pp. To appear in Margaria and Yi, editors,** ***Tools and Algorithms for The Construction and Analysis of Systems: 7th International Conference*****, TACAS '01 Proceedings, LNCS, 2001.**