BRICS RS-98-1 O. Danvy: A Simple Solution to Type Specialization

BRICS

Basic Research in Computer Science

A Simple Solution to Type Specialization

Olivier Danvy

BRICS Report Series RS-98-1

ISSN 0909-0878 January 1998

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/98/1/`

A Simple Solution to Type Specialization

Olivier Danvy

BRICS
Department of Computer Science
University of Aarhus
Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark
E-mail: `danvy@brics.dk`
Home page: `http://www.brics.dk/~danvy`

Abstract. Partial evaluation specializes terms, but traditionally this specialization does not apply to the type of these terms. As a result, specializing, e.g., an interpreter written in a typed language, which requires a "universal" type to encode expressible values, yields residual programs with type tags all over. Neil Jones has stated that getting rid of these type tags was an open problem, despite possible solutions such as Torben Mogensen's "constructor specialization." To solve this problem, John Hughes has proposed a new paradigm for partial evaluation, "Type Specialization," based on type inference instead of being based on symbolic interpretation. Type Specialization is very elegant in principle but it also appears non-trivial in practice.
Stating the problem in terms of types instead of in terms of type encodings suggests a very simple type-directed solution, namely, to use a projection from the universal type to the specific type of the residual program. Standard partial evaluation then yields a residual program without type tags, simply and efficiently.

1 The Problem

1.1 An example

Say that we need to write an evaluator in a typed language, such in Figure 1. To this end we use a "universal" data type encoding all the expressible values. As witnessed by the type of the evaluator, evaluating an expression yields a value of the universal type.

```
- eval (LAM ("x", ADD (VAR "x", LIT 1))) Env.init;
val it = FUN fn : univ
-
```

We can visualize the text of this universal value by using *partial evaluation* [3, 10], i.e., by specializing the evaluator of Figure 1 with respect to the expression above. In that, specializing an interpreter with respect to a program provides

```
datatype exp = LIT of int
             | VAR of string
             | LAM of string * exp
             | APP of exp * exp
             | ADD of exp * exp

datatype univ = INT of int
              | FUN of univ -> univ

exception TypeError

(* eval : exp -> univ Env.env -> univ *)
fun eval (LIT i) env
    = INT i
  | eval (VAR x) env
    = Env.lookup (x, env)
  | eval (LAM (x, e)) env
    = FUN (fn v => eval e (Env.extend (x, v, env)))
  | eval (APP (e0, e1)) env
    = (case (eval e0 env) of
          (FUN f) => f (eval e1 env)
        | _ => raise TypeError)
  | eval (ADD (e1, e2)) env
    = (case (eval e1 env, eval e2 env) of
          (INT i1, INT i2) => INT (i1 + i2)
        | _ => raise TypeError)

signature ENV
= sig
    type 'a env
    exception UndeclaredIdentifier
    val extend : string * 'a * 'a env -> 'a env
    val init : 'a env
    val lookup : string * 'a env -> 'a
  end
```

Fig. 1. An example evaluator in Standard ML

a mechanized solution to the traditional exercise in denotational semantics of exhibiting the denotation of a program [13, Exercises 1 and 2, Chapter 5].

A simple partial evaluator would perform the recursive descent and the environment management of the evaluator, and yield a residual term such as the following one.

```
FUN (fn v => (case (v, INT 1) of
                (INT i1, INT i2) => INT (i1 + i2)
              | _ => raise TypeError))
```

A slightly more enterprising partial evaluator would propagate the constant 1 and fold the corresponding computation, essentially yielding the following residual term.

```
FUN (fn (INT i1) => INT (i1 + 1)
      | _ => raise TypeError)
```

In both cases, the residual program is cluttered with the type tags `FUN` and `INT`.

1.2 The problem

Obtaining a residual program without type tags by specializing an interpreter expressed in a typed language has been stated as an open problem for about ten years now [8, 9]. This problem has become acute with the advent of partial evaluators for typed languages, such as SML-Mix [2].

We note that this problem does not occur for *command* interpreters, which essentially have the functionality `cmd -> sto -> sto`. No matter which command such an interpreter is specialized with respect to, the result is of type `sto`.

The problem only arises for *expression* interpreters, whose codomain *depends* on their domain. Indeed, the type of an expressible value depends on the type of the corresponding source expression.

2 A Sophisticated Solution: "Type Specialization"

Partial evaluation is traditionally performed by non-standard interpretation [3, 10]: during specialization, part of the source term is interpreted, and the rest is reconstructed, yielding a residual term.

Recently, John Hughes has proposed to shift perspective and to perform partial evaluation by non-standard type inference instead of by non-standard interpretation [6]. In doing so, he has achieved both term and type specialization. His new approach has been favorably met in the functional-programming community [7].

The resulting type specialization is very elegant in principle but so far it appears non-trivial in practice. Like all other partial evaluators in their infancy, in its current state, it requires expert source annotations to work. Correspondingly, no efficient implementations seem to exist yet, despite recent progress [14].

3 A Simpler Solution: Projecting from the Universal Type

Let us go back to the example of Section 1.1. There is an obvious embedding/projection between the native types of ML and the universal type `univ`. Noting ε for the embedding and π for the projection, it reads as follows.

$$\begin{cases} \varepsilon_{\text{int}}\, i = \texttt{INT}\ i \\ \varepsilon_{t_1 \to t_2}\, f = \texttt{FUN}\ \lambda v.\varepsilon_{t_2}\,(f\,(\pi_{t_1}\, v)) \\ \\ \pi_{\text{int}}\,(\texttt{INT}\ i) = i \\ \pi_{t_1 \to t_2}\,(\texttt{FUN}\ f) = \lambda v.\pi_{t_2}\,(f\,(\varepsilon_{t_1}\, v)) \end{cases}$$

Thus equipped, we can project the expressible value of Section 1.1 from the universal type to the type of the original expression, i.e., `int -> int`. In doing so, we obtain

1. a value of type `int -> int` by evaluation; and
2. a residual program without type tags by partial evaluation, that reads:

```
fn v => v + 1
```

Our simple solution thus amounts to composing the projection with the interpreter prior to partial evaluation. In effect, the projection specializes the type, and in practice, the partial evaluator specializes the term, including its projection.

4 A Case Study

We have paired the embedding/projection described above with type-directed partial evaluation [4], both in Scheme and in ML. Here is a typical measure in Standard ML of New Jersey, Version 0.93, on a 150Mhz Pentium running Linux. The ML measures are more significant than the Scheme measures because they do not depend on our particular tagged representation of typed values. For lack of an interactive timer, we are not able to report similar measures in Caml [1].

For this measure, we have extended the interpreter of Figure 1 to handle a more substantial language including booleans, conditional expressions, recursive functions, and extra numerical operations.

We repeated the following computations 1000 times. The resulting numbers should thus be divided by 1000. They include garbage collection.

We consider the functional F associated to the factorial function (but using addition instead of multiplication, to avoid numeric overflow).

Term overhead and type overhead: Let m denote the result of applying the interpreter to F. The value m has type `univ`. Applying the fixed point of m to 100 and projecting its result, i.e.,

$$\pi_{\text{int}}\,(\text{fix}_{\texttt{univ}}\, m\,(\varepsilon_{\text{int}}\, 100))$$

(repeated 1000 times) yields 5050 in 4.1 seconds.

Term overhead and type overhead, plus the projection: We then project m, obtaining a value of type `(int -> int) -> int -> int`. Applying the fixed point of this value to 100, i.e.,

$$\mathrm{fix}_{(\mathrm{int}\to\mathrm{int})\to\mathrm{int}\to\mathrm{int}}\,(\pi_{(\mathrm{int}\to\mathrm{int})\to\mathrm{int}\to\mathrm{int}}\,m)\,100$$

(repeated 1000 times) yields 5050 in 4.9 seconds.

The projection slows down the computation by about 20%.

No term overhead, but type overhead: We now consider M, the result of specializing the interpreter with respect to F. The meaning of M has type `univ`. Applying the fixed point of this meaning to 100 and projecting its result, i.e.,

$$\pi_{\mathrm{int}}\,(\mathrm{fix}_{\mathtt{univ}}\,[\![M]\!]\,(\varepsilon_{\mathrm{int}}\,100))$$

(repeated 1000 times) yields 5050 in 0.5 seconds.

No term overhead and no type overhead: We now consider M', the result of specializing the projected interpreter with respect to F. The meaning of M' is of type `(int -> int) -> int -> int`. Applying the fixed point of this meaning to 100, i.e.,

$$\mathrm{fix}_{(\mathrm{int}\to\mathrm{int})\to\mathrm{int}\to\mathrm{int}}\,[\![M']\!]\,100$$

(repeated 1000 times) yields 5050 in 0.3 seconds.

Overhead of type-directed partial evaluation: specializing the projected denotation of F (repeated 1000 times) takes 0.4 seconds.

Analysis: Specializing the interpreter removes the term overhead: the residual computation is about 88% faster than the original one. Specializing the projected interpreter removes both the term and the type overhead: the residual computation is about 94% faster than the original one with the projection and about 92% faster than the original one without the projection. Finally, the type overhead slows down the residual code by about 67%.

As for the cost of specialization, it is immediately amortized since the time spent running the residual code plus the time spent for partial evaluation is less than the time spent running the source code.

Using Standard ML, we were not able to measure the time spent compiling the residual code. However, we could estimate it using Caml and Chez Scheme. Our Caml implementation combines type-directed partial evaluation and run-time code generation [1], and Chez Scheme offers run-time code generation through `eval` [11]. In both cases, the time spent specializing, compiling the residual code, and running it is vastly inferior to the time spent running the source code.

```
local datatype 'a Fix = FIX of 'a Fix -> 'a
in fun fix_univ (FUN f)
       = let fun g (FIX x)
                 = f (FUN (fn a => let val (FUN h) = x (FIX x)
                                   in h a
                                   end))
         in g (FIX g)
         end
end
```

Fig. 2. Universal fixed-point operator

```
structure Ep
= struct
    datatype 'a ep
        = EP of ('a -> univ) * (univ -> 'a)

    val ep_int
        = EP (fn e => (INT e), fn (INT e) => e)

    fun ep_fun (EP (embed1, project1), EP (embed2, project2))
        = EP (fn f => FUN (fn x => embed2 (f (project1 x))),
              fn (FUN f) => fn x => project2 (f (embed1 x)))
  end

fun ts (Ep.EP (embed, project)) x = project x
fun tg (Ep.EP (embed, project)) x = embed x

val int = Ep.ep_int
infixr 5 -->; val op --> = Ep.ep_fun;
```

Fig. 3. Embeddings and projections in ML (after Andrzej Filinski and Zhe Yang)

5 Implementation

We have specified and implemented the embedding/projection of Section 3 to make it handle unit, booleans, products, disjoint sums, lists, and constructor specialization [12]. Except for constructor specialization, the embedding/projection is trivial since ML supports unit, booleans, products, disjoint sums and lists. Constructor specialization is handled with an encoding in ML. Other type constructs that are not native in ML would be handled similarly, i.e., with an encoding. Recursion is handled through a fixed-point operator (see Figure 2).

It is not completely immediate to implement embedding/projection pairs in ML. We did it using a programming technique due to Andrzej Filinski (personal communication, Spring 1995) and Zhe Yang (personal communication, Spring 1996) [15], originally developed to implement type-directed partial evaluation in ML. The technique works in two steps:

1. defining a polymorphic constructor of embedding/projection pairs for each type constructor; and
2. constructing the corresponding pair, following the inductive structure of the type.

Given such a pair, one can achieve type specialization with its projection part, and type generalization with its embedding part, as defined in Figure 3 and illustrated in the following interactive session.

```
- tg (int --> int) (fn x => x + 1);
val it = FUN fn : univ
- ts (int --> int) (FUN (fn (INT x) => INT (x + 1)));
std_in:22.24-22.48 Warning: match nonexhaustive
          INT x => ...

val it = fn : int -> int
-
```

And along the same lines, one can add a polymorphic component to `univ`.

6 An Improvement

With its pairs of type-indexed functions reify and reflect [4], type-directed partial evaluation is defined very similarly to the embedding/projection pairs considered in this article. It is therefore tempting to compose them, to specialize the interpreter at the same time as we are projecting it.

The results are two-level versions of the embedding/projection pairs:

$$\begin{cases} \varepsilon_{t_1 \to t_2}\, \mathtt{f} = \mathtt{FUN}\ \lambda v.\varepsilon_{t_2}\,(\mathtt{APP\ (f,}\ \pi_{t_1}\, v\mathtt{)}) \\ \pi_{\text{int}}\,(\mathtt{INT}\ i) = \mathtt{LIT}\ i \\ \pi_{t_1 \to t_2}\,(\mathtt{FUN}\ f) = \mathtt{LAM\ (x,}\ \pi_{t_2}\,(f\,(\varepsilon_{t_1}\,(\mathtt{VAR\ x}))\mathtt{))} \qquad \text{where } \mathtt{x} \text{ is fresh.} \end{cases}$$

Each projection now maps a universal value into the text of its normal form (if it exists), and each embedding maps a text into the corresponding universal value. As usual in offline type-directed partial evaluation, one cannot embed a dynamic integer. (Base types can only occur positively in the source type [5].)

The following ML session illustrates how to residualize universal values, using the two-level embedding/projection pairs defined just above.

```
- residualize (a --> a) (FUN (fn x => x));
val it = LAM ("x1",VAR "x1") : exp
- residualize ((int --> a) --> a) (FUN (fn (FUN f) => f (INT 42)));
std_in:53.14-53.37 Warning: match nonexhaustive
          FUN f => ...

val it = LAM ("x1",APP (VAR "x1",LIT 42)) : exp
- residualize (a --> int) (FUN (fn x => INT (1+1)));
val it = LAM ("x1",LIT 2) : exp
-
```

The last interaction illustrates the normalization effect of residualization (`1+1` was calculated at residualization time).

7 Conclusion and Issues

Traditionally, partial evaluators have mostly been developed for untyped languages, where type specialization is not a concern. Type specialization, however, appears to be a real issue for typed languages [9]. The point is that to be satisfactory, partial evaluation of typed programs must specialize both terms and types, and traditional partial evaluators specialize only terms. Against this shortcoming of traditional partial evaluation, John Hughes has proposed an elegant new paradigm to specialize both terms and types [6, 7]. We suggest the simpler and more conservative solution of (1) using a projection to achieve type specialization, and (2) reusing traditional partial evaluation to carry out the corresponding term specialization. This solution requires no other insight than knowing the type of the source program and, in the case of definitional interpreter, its associated type transformer.[1] In combination with type-directed partial evaluation, it also appears to be very efficient in practice.

Given a statically typed functional language such as ML or Haskell, and using Andrzej Filinski and Zhe Yang's inductive technique, it is very simple to write embedding/projection pairs. This simplicity, plus the fact that, as outlined in Section 6, they mesh very well with type-directed partial evaluation, counterbalance the fact that one needs to write such pairs for every new universal type one encounters.

[1] For example, the type transformation associated to an interpreter in direct style is the identity transformation, the type transformer associated to an interpreter in continuation style is the CPS transformation, etc.

As several anonymous referees pointed out, using embedding/projection pairs is not as general as John Hughes's approach. It however has the advantage of being directly usable since it builds on all the existing partial-evaluation technology.

But getting back to the central issue of type specialization, i.e., specializing both terms and types, and how it arose, i.e., to specialize expression interpreters, the author is struck by the fact that such interpreters are dependently typed. Therefore, he conjectures that either the partial-evaluation technology we are building will prove useful to implement dependently typed programs, or that conversely the wealth of work on dependent types will provide us with guidelines for partially evaluating dependently typed programs – probably a little of both.

Acknowledgements

This work is supported by BRICS (Basic Research in Computer Science, Centre of the Danish National Research Foundation).

Thanks to Neil D. Jones for letting me present this simple solution to type specialization at DIKU in January 1998, and to the whole TOPPS group for the ensuing lively discussion. Thanks also to the organizers of the CLICS lunch, at BRICS, for letting me air this idea at an early stage.

I am grateful to Belmina Dzafic, Karoline Malmkjær, and Zhe Yang for their benevolent ears in the fall of 1997, and to the anonymous referees for their pertinent reviews.

And last but not least, many thanks are due to Andrzej Filinski and Zhe Yang for their beautiful programming technique!

References

1. Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation (preliminary version). Technical Report BRICS RS-97-43, Department of Computer Science, University of Aarhus, Aarhus, Denmark, October 1997. To appear in the proceedings of TIC'98.
2. Lars Birkedal and Morten Welinder. Partial evaluation of Standard ML. Master's thesis, DIKU, Computer Science Department, University of Copenhagen, August 1993. DIKU Rapport 93/22.
3. Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In Susan L. Graham, editor, *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 493–501, Charleston, South Carolina, January 1993. ACM Press.
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. Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998.

World Scientific. Extended version available as the technical report BRICS RS-97-53.
6. John Hughes. Type specialisation for the lambda calculus; or, a new paradigm for partial evaluation based on type inference. In Olivier Danvy, Robert Glück, and Peter Thiemann, editors, *Partial Evaluation*, number 1110 in Lecture Notes in Computer Science, Dagstuhl, Germany, February 1996. Springer-Verlag.
7. John Hughes. An introduction to program specialisation by type inference. In *Functional Programming*, Glasgow University, July 1996. Published electronically.
8. Neil D. Jones. Challenging problems in partial evaluation and mixed computation. In Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 1–14. North-Holland, 1988.
9. Neil D. Jones. Relations among type specialization, supercompilation and logic program specialization. In Hugh Glaser and Herbert Kuchen, editors, *Ninth International Symposium on Programming Language Implementation and Logic Programming*, number 1292 in Lecture Notes in Computer Science, Southampton, UK, September 1997. Invited talk.
10. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.
11. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *LISP and Symbolic Computation*, 1998. To appear.
12. Torben Æ. Mogensen. Constructor specialization. In David A. Schmidt, editor, *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–32, Copenhagen, Denmark, June 1993. ACM Press.
13. David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.
14. Per Sjörs. Type specialization of a subset of Haskell. Master's thesis, Chalmers University, June 1997.
15. Zhe Yang. Encoding types in ML-like languages. Draft, Department of Computer Science, New York University, April 1998.

Recent BRICS Report Series Publications

RS-98-1 **Olivier Danvy.** ***A Simple Solution to Type Specialization*****. January 1998. 10 pp. Appears in Larsen, Skyum and Winskel, editors,** ***25th International Colloquium on Automata, Languages, and Programming*****, ICALP '98 Proceedings, LNCS 1443, 1998, pages 908–917.**

RS-97-53 **Olivier Danvy.** ***Online Type-Directed Partial Evaluation*****. December 1997. 31 pp. Extended version of an article appearing in** ***Third Fuji International Symposium on Functional and Logic Programming*****, FLOPS '98 Proceedings (Kyoto, Japan, April 2–4, 1998), pages 271–295, World Scientific, 1998.**

RS-97-52 **Paola Quaglia.** ***On the Finitary Characterization of π-Congruences*****. December 1997. 59 pp.**

RS-97-51 **James McKinna and Robert Pollack.** ***Some Lambda Calculus and Type Theory Formalized*****. December 1997. 43 pp Appears in** ***Journal of Automated Reasoning*****, 23(3–4):373–409, 1999.**

RS-97-50 **Ivan B. Damgård and Birgit Pfitzmann.** ***Sequential Iteration of Interactive Arguments and an Efficient Zero-Knowledge Argument for NP*****. December 1997. 19 pp. Appears in Larsen, Skyum and Winskel, editors,** ***25th International Colloquium on Automata, Languages, and Programming*****, ICALP '98 Proceedings, LNCS 1443, 1998, pages 772–783.**

RS-97-49 **Peter D. Mosses.** ***CASL for ASF+SDF Users*****. December 1997. 22 pp. Appears in Sellink, Editor,** ***2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing*****, ASF+SDF '97 Proceedings, Springer-Verlag, 1997.**

RS-97-48 **Peter D. Mosses.** ***CoFI: The Common Framework Initiative for Algebraic Specification and Development*****. December 1997. 24 pp. Appears in Bidoit and Dauchet, editors,** ***Theory and Practice of Software Development: 7th International Joint Conference CAAP/FASE*****, TAPSOFT '97 Proceedings, LNCS 1214, 1997, pages 115–137.**