BRICS

Basic Research in Computer Science

Strong Normalization by Type-Directed Partial Evaluation and Run-Time Code Generation

(Preliminary Version)

Vincent Balat
Olivier Danvy

BRICS Report Series **RS-97-43**

ISSN 0909-0878 **December 1997**

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/97/43/`

Strong Normalization by Type-Directed Partial Evaluation and Run-Time Code Generation (preliminary version) *

Vincent Balat
Dépt. d'Informatique
ENS de Cachan †

Olivier Danvy
BRICS ‡
Dept. of Computer Science
University of Aarhus §

December 1997

Abstract

We investigate the synergy between type-directed partial evaluation and run-time code generation for the Caml dialect of ML. Type-directed partial evaluation maps simply typed, closed Caml values to a representation of their long $\beta\eta$-normal form. Caml uses a virtual machine and has the capability to load byte code at run time. Representing the long $\beta\eta$-normal forms as byte code gives us the ability to strongly normalize higher-order values (i.e., weak head normal forms in ML), to compile the resulting strong normal forms into byte code, and to load this byte code all in one go, at run time.

We conclude this note with a preview of our current work on scaling up strong normalization by run-time code generation to the Caml module language.

---

*To appear in the preliminary proceedings of TIC'98.

†61, avenue du Président Wilson, F-94230 Cachan Cedex, France.
E-mail: `balat@rip.ens-cachan.fr`
This work was carried out at BRICS during the summer of 1997.
‡Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
Home page: `http://www.brics.dk`
§Ny Munkegade, Building 540, DK-8000 Aarhus C, Denmark.
E-mail: `danvy@brics.dk`

1 Introduction

1.1 Motivation

Strong normalization: Suppose one is given a strongly normalizable (closed) λ-term. How does one normalize this term? Typically one parses it into an abstract-syntax tree, one writes a strong normalizer over abstract-syntax trees, and one translates (unparses) the resulting normal form into whichever desired format (e.g., LaTeX).

A solution in ML: ML, like all functional languages, provides a convenient format for representing λ-terms: as an ML expression. Suppose thus that we are given a strongly normalizable ML expression. How do we normalize it? Type-directed partial evaluation [6] offers an efficient alternative to writing a parser to represent this ML expression as an ML data structure representing its abstract-syntax tree, writing a strong normalizer operating over this abstract-syntax tree, and unparsing the resulting normal form into an ML expression. Instead, the ML evaluator maps this ML expression into an ML value, and the type-directed partial evaluator maps this ML value into the abstract-syntax tree of its normal form. We can then either evaluate this abstract-syntax tree (for immediate use) or unparse it (for later use).

Motivation: Type-directed partial evaluation entrusts the underlying programming language with all the mechanisms of binding and substitution that are associated with normalization. Higher-Order Abstract Syntax [17] shares the same motivation, albeit in a Logical Framework instead of in a functional setting.

Goal: Type-directed partial evaluation, as it is, maps an ML value into the text of its normal form. We want instead to map it into the corresponding ML value — and we want to do that in a lighter way than by invoking either an interpreter or the whole compiler, after normalization.

An integrated solution in Objective Caml: Objective Caml [15] is a byte-code implementation of a dialect of ML. This suggests us to represent normal forms as byte code, and to load this byte code at run time for both immediate and later use.

```
module ChurchNumbers
= struct
    let cz s z = z
    let cs n s z = n s (s z)

    let rec n2cn n = if n=0 then cz else cs (n2cn (n-1))
    let cn2n n = n (fun i -> i+1) 0
  end
```

Figure 1: Church numbers

1.2 Contribution

We report our experiment of integrating type-directed partial evaluation within Caml, which in effect yields strong normalization by run-time code generation. We list below what we had to do to achieve this integration:

- we wrote several type-directed partial evaluators in Caml, in various styles and with various properties (listed below);
- we wrote a dedicated translator from normal forms to byte code;
- this required us to find the necessary (hidden) resources in the Caml implementation and recompile the system to make them available, in effect obtaining a more open implementation. These resources are mainly the representation of types, the representation of byte code, and the ability to load byte code at run time.

1.3 Non-contribution

Even though it is primarily inspired by theory, our work is experimental. Indeed, neither the OCaml compiler nor the OCaml virtual machine are formalized. We therefore have not formalized our byte-code translator either. As for type-directed partial evaluation, only its call-by-name version has been formalized so far [1, 2, 5].

In that sense our work is experimental: we want to investigate the synergy between type-directed partial evaluation and run-time code generation for OCaml.

1.4 An example: Church numbers

Let us illustrate strong normalization by run-time code generation to optimize a computation over Church numbers, which we define in Figure 1. The module `ChurchNumbers` defines zero (`cz`), the successor function (`cs`), and two conversion functions to and from ML numbers and Church numbers.

For example, we can convert the ML number 5 to a Church number, increment it, and convert the result back to ML as follows:

```
# ChurchNumbers.cn2n(ChurchNumbers.cs (ChurchNumbers.n2cn 5));;
- : int = 6
#
```

Thus equipped, let us define the function incrementing its argument with 1000:

```
# let cs1000 m = ChurchNumbers.n2cn 1000 ChurchNumbers.cs m;;
val cs1000 : (('a -> 'a) -> 'a -> 'b) -> ('a -> 'a) -> 'a -> 'b
           = <fun>
# ChurchNumbers.cn2n(cs1000 (ChurchNumbers.cz));;
- : int = 1000
#
```

If it were not for ML's weak-normalization strategy, 1000 β-reductions could be realized at definition time. We strongly normalize the value denoted by `cs1000` by invoking our function `nip` (for "Normalize In Place") on the name of the identifier `cs1000`:

```
# nip "cs1000";;
- : unit = ()
#
```

Now `cs1000` denotes the strongly normalized value, as reflected by its execution time: applying `cs1000` to the Church number 0 is 4800 times faster now. Depending on the version of the type-directed partial evaluator, normalization takes between 0.1 and 18 seconds. In this example, `cs1000` then needs to be applied between 5 and 1000 times to amortize the cost of normalization.

1.5 Overview

The rest of this article is organized as follows. We first review type-directed partial evaluation (Section 2), independently of run-time code generation, and with two simple examples: the Hilbert combinators and Church numbers. We then describe run-time code generation in OCaml (Section 3). Putting them together, we report the measures we have collected (Section 4) and we assess the overall system (Section 5). The Caml implementation of modules suggests a very simple extension of our system to handling both first-order and higher-order modules, and we describe this extension in Section 6. After reviewing related work (Section 7), we conclude.

2 Type-Directed Partial Evaluation

Type-directed partial evaluation strongly normalizes closed values of parametric type, by *two-level η-expansion* [6, 8]. Let us take two concrete examples, a simple one first, and a second one with Church numbers. We represent residual lambda-terms with the data type of Figure 2.

2.1 The Hilbert combinators

As is well-known, the identity combinator I can be defined with the Hilbert combinators S and K. This is often illustrated in ML with the functions `cS` and `cK` defined in Figure 3:

```
# let cI x = SK.cS SK.cK SK.cK x;;
val cI : 'a -> 'a = <fun>
# cI 42;;
- : int = 42
```

It is the point of type-directed partial evaluation that can visualize the text of `cI` by two-level η-expansion. In the present case, all we need is to η-expand `cI` with a dynamic introduction rule (the construction of a residual lambda-abstraction) and a static elimination rule (an ML application):

```
# let ee_id f = Lam("x", f (Var "x"));;
val ee_id : (exp -> exp) -> exp = <fun>
# ee_id (SK.cS SK.cK SK.cK);;
- : exp = Lam ("x", Var "x")
#
```

where in the definition of `ee_id`, `x` is fresh. The result of applying `ee_id` to the ML identity function is its text in normal form.

```
type exp = Var of string
         | Lam of string * exp
         | App of exp * exp
```

Figure 2: Abstract syntax of the λ-calculus

```
module type SK_sig
= sig
    val cS : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
    val cK : 'a -> 'b -> 'a
  end

module SK : SK_sig
= struct
    let cS f g x = f x (g x)
    let cK a b = a
  end
```

Figure 3: Hilbert's Combinatory Logic basis

2.2 Church numbers

Let us play the same game with Church numbers. The type of a Church number is

```
('a -> 'a) -> 'a -> 'a
```

Since it is composed with three arrows, we need to η-expand it three times. Since the two outer arrows occur positively, we η-expand a Church number `cn` with two dynamic introduction rules and two static elimination rules:

```
Lam("s", Lam("z", cn (...(Var "s")...)  (Var "z")))
```

where `s` and `z` are fresh.

Since the inner arrow occurs negatively, we η-expand the corresponding variable `s` with one static introduction rule (an ML abstraction) and one dynamic elimination rule (the construction of a residual application):

```
fun v -> App(Var "s", v)
```

$$
\begin{array}{rcl}
\langle\text{type}\rangle & ::= & \langle\text{covariant-type}\rangle \\[1ex]
\langle\text{covariant-type}\rangle & ::= & \langle\text{base-type}\rangle \\
 & | & \texttt{variable} \\
 & | & \langle\text{contravariant-type}\rangle \texttt{ -> } \langle\text{covariant-type}\rangle \\
 & | & \langle\text{covariant-type}\rangle \texttt{ * } \ldots \texttt{ * } \langle\text{covariant-type}\rangle \\[1ex]
\langle\text{contravariant-type}\rangle & ::= & \texttt{bool} \\
 & | & \texttt{variable} \\
 & | & \langle\text{covariant-type}\rangle \texttt{ -> } \langle\text{contravariant-type}\rangle \\
 & | & \langle\text{contravariant-type}\rangle \texttt{ * } \ldots \texttt{ * } \langle\text{contravariant-type}\rangle \\[1ex]
\langle\text{base-type}\rangle & ::= & \texttt{unit} \mid \texttt{int} \mid \texttt{float} \mid \texttt{bool} \mid \texttt{string}
\end{array}
$$

Figure 4: Abstract syntax of types

The result reads as follows:

```
# let ee_cn cn
      = Lam("s",
            Lam("z",
                cn (fun v -> App(Var "s", v)) (Var "z")));;
val ee_cn : ((exp -> exp) -> exp -> exp) -> exp = <fun>
#
```

We are now equipped to visualize the normal form of a Church number, e.g., 3:

```
# ee_cn (ChurchNumbers.n2cn 3);;
- : exp = Lam("s",
              Lam("z",
                  App(Var "s",
                          App(Var "s",
                              App(Var "s", Var "z")))))
#
```

The result of applying ee_cn to the ML Church number 3 is the text of this Church number in normal form.

2.3 Summary and conclusion

We have illustrated type-directed partial evaluation in ML with two very simple examples: the Hilbert combinators and Church numbers. We have defined them in ML and we have constructed the text of their normal form, by two-level η-expansion.

Type-directed partial evaluation directly constructs two-level η-redices, given a representation of the type of the value to normalize. It also handles more types, such as base types (in restricted position), and can interpret function types as having a computational effect (in which case it inserts a residual let expression, using continuations). Figure 4 displays our grammar of admissible types.

We therefore implemented several type-directed partial evaluators:

- inserting or not inserting let expressions; and
- in a purely functional way, i.e., implementing two-level eta-expansion directly in ML, using Filinski's strategy,[1] or with an explicit representation of two-level terms as the abstract-syntax tree of an ML expression (which is then compiled).

In the following section, instead of constructing a normal form as an abstract-syntax tree, we construct byte code and load it in place, thereby obtaining the effect of strong normalization by type-directed partial evaluation and run-time code generation.

3 Run-Time Code Generation

We therefore have written a translator mapping a term in long $\beta\eta$-normal form into equivalent byte code for the OCaml virtual machine. And we load this byte code and update in place the value we have normalized.

3.1 Generating byte code

We do not generate byte code by calling the Caml compiler on the text of the normal forms. The language of normal forms is a tiny subset of ML, and therefore we represent it with a dedicated abstract syntax. Since normal forms are well typed, we also shortcut the type-checking phase of the compiler. Finally, we choose not to use the resident byte-code generator: instead, we use our own translator from normal forms to byte code.

[1] Personal communication, spring 1995.

3.2 Loading byte code

For this we need to access OCaml's byte-code loader, which required us to open its implementation. We have thus added more entry points in some of the modules that are available at the user level (i.e., Caml's toplevel). We have also made several interfaces available, by copying them in the OCaml libraries.

We essentially needed access to functions for loading byte code, and access to the current environment and its associated access functions. As a side benefit, our user does not need to specify the type of the value to optimize, since we can retrieve this information in the environment.

3.3 Updating in situ

Finally, being given the name of a variable holding a value to optimize, and being able to find its type in the environment, nothing prevents us to update the binding of this variable with the optimized value — which we do. We illustrated the whole process in Section 1.4, by

- defining a variable `cs1000` denoting 1000 compositions of Church's successor function, and
- normalizing it in place with our function `nip`.

4 Applications

We have tested our system with traditional partial-evaluation examples, the biggest of which are definitional interpreters for programming languages. The results are consistent with the traditional results reported in the partial-evaluation literature [13]: the user's mileage may vary, depending (in the present case) on how much strong normalization is hindered by ML's weak-normalization strategy.

The definitional interpreters we have considered are traditional in partial evaluation: they range from a simple while language [3] to an Algol-like language with subtyping and recursion [9]. Our interpreters are written in Caml. Some use continuation-passing style (CPS), and the others direct style. In the definitional interpreters, iteration and recursion are handled with fixed-point operators.

All our examples clearly exhibit a speedup after normalization. The specialized version of an interpreter with respect to a program, for example, is typically 2.5 times faster after normalization. On some other examples (e.g., Section 1.4), the residual programs are several thousand times faster than the (unnormalized) source program.

The computational resources mobilized by type-directed partial evaluation vary wildly, depending on the source program. For example, specializing a direct-style interpreter with respect to a 10000-lines program takes 45 seconds and requires about 170 runs to be amortized. Specializing a CPS interpreter with respect to a 500-lines program, on the other hand, takes 20 minutes. We believe that this low performance is due to an inefficient handling of CPS in OCaml. Essentially the same implementation takes a handful of seconds in Chez Scheme for a 1000-lines program, with less than 0.5 seconds for type-directed partial evaluation proper, and with a fairly small difference if the interpreter is in direct style or in CPS.

We also experimented with the resident OCaml byte-code generator, which is slower by a factor of at least 3 than our dedicated byte-code generator. This difference demonstrates that using a special-purpose byte-code generator for normal forms is a significant optimization.

5 Assessment

Although so far we are its only users, we believe that our system works reasonably well. In fact, we are in the process of writing a users's manual.

Our main problem at this point is the same as for any other partial evaluator: speedups are completely problem-dependent. In contrast with most other partial evaluators, however, we can quantify this statement: because (at least in its pure form) type-directed partial evaluation strongly normalizes its argument, we can state that it provides all the (strong) normalization steps that are hindered by ML's weak-normalization strategy.

Our secondary problem is efficiency: because OCaml is a byte-code implementation, it is inherently slower than a native code implementation such as Chez Scheme [11], which is our reference implementation. Therefore our benchmarks in OCaml are typically measured in dozens of seconds whereas they are measured in very few seconds in Chez Scheme.[2] Efficiency becomes even more of a problem for the continuation-based version of the

[2]For comparison, an interpreter-based and optimized implementation of type-directed partial evaluation in ML consistently performs between 1000 and 10000 times slower than the implementation in Chez Scheme [18]. The point here is not byte code vs. native code, but interpreted code vs. compiled code.

type-directed partial evaluator: whereas Chez Scheme represents continuations very efficiently [12], that is not the case at all for OCaml. Conversely, the continuation-based partial evaluator yields perceptibly better residual programs (e.g., without code duplication because of let insertion).

Caveat: If our system is given a diverging source program, it diverges as well. In that it is resource-unbounded [7, 10].

6 Towards Modular Type-Directed Partial Evaluation

In a certain sense, ML's higher-order modules are essentially the simply typed lambda-calculus laid on top of first-order modules ("structures") [16]. Looking under the hood, that is precisely how they are implemented. This suggests us to extend our implementation to part of the Caml module language.

Enabling technology: After type-checking, first-order modules ("structures") are handled as tuples and higher-order modules ("functors") are handled as higher-order functions. Besides, enough typing information is held in the environment to be able to reconstruct their type. Put together, these two observations make it possible for us to reuse most of our existing implementation.

Achievements and limitations: We handle a subset of the Caml module language, excluding polymorphism and sharing constraints.

An example: typed Combinatory Logic. Let us build on the example of Section 2.1. We have located the definition of the Hilbert combinators in a module defining our standard Combinatory Logic basis (see Figure 3). We then define an alternative basis in another module, in terms of the first one (see Figure 5). Because of ML's weak-normalization strategy, using the alternative basis incurs an overhead. We can eliminate this overhead by normalizing in place the alternative basis:

```
# nip_module "BCWK";;
- : unit = ()
#
```

```
module type BCWK_sig
= sig
    val cB : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
    val cC : ('a -> 'b -> 'c) -> 'b -> 'a -> 'c
    val cW : ('a -> 'a -> 'b) -> 'a -> 'b
    val cK : 'a -> 'b -> 'a
  end
module BCWK : BCWK_sig
= struct
    open SK
    let cB f g x = cS (cK cS) cK f g x
    let cC f x y = cS (cS (cK (cS (cK cS) cK)) cS) (cK cK) f x y
    let cW f x = cS cS (cK (cS cK cK)) f x
    let cK = cK
  end
```

Figure 5: A Combinatory Logic basis of regular combinators

What happens here is that the identifier `BCWK` denotes a tuple with four entries, each of which we already know how to process. Given the name of this identifier, the implementation

1. locates it in the Caml environment;
2. accesses its type;
3. constructs the simple type of a tuple of four elements;
4. strongly normalizes it, using type-directed partial evaluation;
5. translates it into byte code, and loads it;
6. updates in place the environment to make the identifier `BCWK` denote the generated code.

7 Related Work

Partial evaluation is traditionally defined as a source-to-source program transformation [4, 13]. Type-directed partial evaluation departs from that tradition in that it is a compiled-to-source program transformation. Run-time code generation completes the picture by providing a source-to-compiled transformation at run time. It is thus a natural idea to compose both, and this has been done in two settings, using offline partial-evaluation techniques:

For imperative languages: the Compose research group at Rennes is doing run-time code generation for stock languages such as C, C^{++}, and Java.

For functional languages: Sperber and Thiemann have paired a traditional, syntax-directed partial evaluator and a run-time code generator for a byte-code implementation of Scheme [19].

Both settings use binding-time analysis. Sperber and Thiemann's work is the most closely related to ours, even though their partial evaluator is syntax-directed instead of type-directed and though they consider an untyped and module-less language (Scheme) instead of a typed and modular one (ML). A remarkable aspect of their work, and one our implementation so far has failed to achieve, is that they deforest the intermediate representation of the specialized program, i.e., their partial evaluator directly generates byte code.

Alternative approaches to partial evaluation and run-time code generation include Leone and Lee's Fabius system [14], which only handles "staged" first-order ML programs but generates actual assembly code very efficiently.

8 Conclusion and Issues

We have obtained strong normalization in ML by pairing type-directed partial evaluation and run-time code generation. We have implemented a system in Objective Caml, whose byte code made it possible to remain portable. The system can be used in any situation where strong normalization could be of benefit. For example, we are considering to apply it for cut elimination in formal proofs, in a proof assistant.

We are in the process of extending our implementation for a subset of the Caml module language. This extension relies on the run-time treatment of structures and of functors, which are represented as tuples and as higher-order functions. Therefore, in a pre-pass, we assemble type information about the module to normalize (be it first order or higher order), we coerce it into simply typed tuple and function constructions, and we then reuse our earlier implementation.

The practical limitations are the same as for offline type-directed partial evaluation, i.e., source programs must be explicitly factored prior to specialization. The module language, however, appears to be a pleasant support for expressing this factorization.

Acknowledgements

To Xavier Leroy for supplying us with a version of `call/cc` for OCaml.

References

[1] Ulrich Berger. Program extraction from normalization proofs. In M. Bezem and J. F. Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993.

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

[3] Charles Consel and Olivier Danvy. Static and dynamic semantics processing. In Robert (Corky) Cartwright, editor, *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 14–24, Orlando, Florida, January 1991. ACM Press.

[4] 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.

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

[6] 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.

[7] Olivier Danvy, Nevin C. Heintze, and Karoline Malmkjær. Resource-bounded partial evaluation. *ACM Computing Surveys*, 28(2):329–332, June 1996.

[8] Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *LISP and Symbolic Computation*, 8(3):209–227, 1995.

[9] Olivier Danvy and René Vestergaard. Semantics-based compiling: A case study in type-directed partial evaluation. In Herbert Kuchen and Doaitse Swierstra, editors, *Eighth International Symposium on Programming Language Implementation and Logic Programming*, number 1140 in Lecture Notes in Computer Science, pages 182–197, Aachen, Germany, September 1996. Extended version available as the technical report BRICS-RS-96-13.

[10] Saumya Debray. Resource-bounded partial evaluation. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192, Amsterdam, The Netherlands, June 1997. ACM Press.

[11] R. Kent Dybvig. *The Scheme Programming Language*. Prentice-Hall, 1987.

[12] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[13] 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.

[14] Mark Leone and Peter Lee. Lightweight run-time code generation. In Peter Sestoft and Harald Søndergaard, editors, *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, Technical Report 94/9, University of Melbourne, Australia, pages 97–106, Orlando, Florida, June 1994.

[15] Xavier Leroy. *The Objective Caml system, release 1.05*. INRIA, Rocquencourt, France, 1997.

[16] David B. MacQueen. Modules for Standard ML. In Guy L. Steele Jr., editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 198–207, Austin, Texas, August 1984.

[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] Tim Sheard. A type-directed, on-line, partial evaluator for a polymorphic language. In Charles Consel, editor, *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–35, Amsterdam, The Netherlands, June 1997. ACM Press.

[19] Michael Sperber and Peter Thiemann. Two for the price of one: composing partial evaluation and compilation. In Ron K. Cytron, editor, *Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 32, No 5, pages 215–225, Las Vegas, Nevada, June 1997. ACM Press.

Recent BRICS Report Series Publications

RS-97-43 Vincent Balat and Olivier Danvy. *Strong Normalization by Type-Directed Partial Evaluation and Run-Time Code Generation (Preliminary Version)*. December 1997. 16 pp.

RS-97-42 Ulrich Kohlenbach. *On the No-Counterexample Interpretation*. December 1997. 26 pp.

RS-97-41 Jon G. Riecke and Anders B. Sandholm. *A Relational Account of Call-by-Value Sequentiality*. December 1997. 24 pp. Appears in *Twelfth Annual IEEE Symposium on Logic in Computer Science*, LICS '97 Proceedings, pages 258–267.

RS-97-40 Harry Buhrman, Richard Cleve, and Wim van Dam. *Quantum Entanglement and Communication Complexity*. December 1997. 14 pp.

RS-97-39 Ian Stark. *Names, Equations, Relations: Practical Ways to Reason about 'new'*. December 1997. ii+33 pp. This supersedes the earlier BRICS Report RS-96-31. It also expands on the paper presented in Groote and Hindley, editors, *Typed Lambda Calculi and Applications: 3rd International Conference*, TLCA '97 Proceedings, LNCS 1210, 1997, pages 336–353.

RS-97-38 Michał Hańćkowiak, Michał Karoński, and Alessandro Panconesi. *On the Distributed Complexity of Computing Maximal Matchings*. December 1997. 16 pp. To appear in *The Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98.

RS-97-37 David A. Grable and Alessandro Panconesi. *Fast Distributed Algorithms for Brooks-Vizing Colourings (Extended Abstract)*. December 1997. 20 pp. To appear in *The Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '98.

RS-97-36 Thomas Troels Hildebrandt, Prakash Panangaden, and Glynn Winskel. *Relational Semantics of Non-Deterministic Dataflow*. December 1997. 21 pp.

RS-97-35 Gian Luca Cattani, Marcelo P. Fiore, and Glynn Winskel. *A Theory of Recursive Domains with Applications to Concurrency*. December 1997. ii+23 pp. A revised version will appear in the Proceedings of LICS'98.