Basic Research in Computer Science

BRICS RS-98-13 Danvy & Rhiger: Compiling Actions by Partial Evaluation, Revisited

Compiling Actions by Partial Evaluation, Revisited

Olivier Danvy
Morten Rhiger

BRICS Report Series RS-98-13
ISSN 0909-0878 June 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/98/13/`

Compiling Actions by Partial Evaluation, Revisited

Olivier Danvy and Morten Rhiger

BRICS [*]

Department of Computer Science

University of Aarhus [†]

April 6, 1998 (updated June 30, 1998)

Abstract

We revisit Bondorf and Palsberg's compilation of actions using the offline syntax-directed partial evaluator Similix (FPCA'93, JFP'96), and we compare it in detail with using an online type-directed partial evaluator. In contrast to Similix, our type-directed partial evaluator is idempotent and requires no "binding-time improvements." It also appears to consume about 7 times less space and to be about 28 times faster than Similix, and to yield residual programs that are perceptibly more efficient than those generated by Similix.

[*]Basic Research in Computer Science,
Centre of the Danish National Research Foundation.
Home page: `http://www.brics.dk`

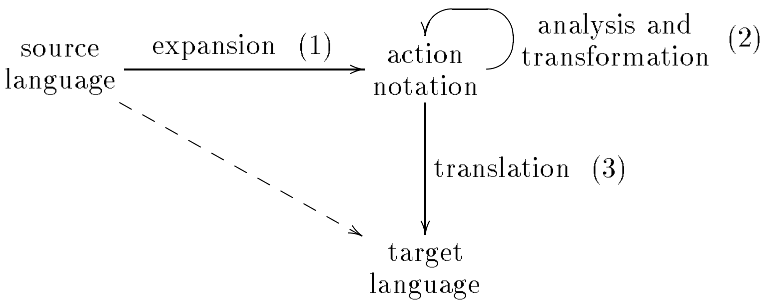
[†]Building 540, Ny Munkegade, DK-8000 Aarhus C, Denmark.
E-mail: {`danvy,mrhiger`}`@brics.dk`

Contents

1 Introduction

The first Futamura projection is probably one of the most celebrated applications of partial evaluation: specializing an interpreter [for a defined language and written in a defining language] with respect to a program yields the effect of compiling this program from the defined language to the defining language [6, 11, 12]. Along with its cousin the second Futamura projection, which requires a self-applicable partial evaluator, the first Futamura projection has been applied to many interpreters embodying a number of programming-language paradigms. Both compiling by specializing an interpreter and the result of this compilation (i.e., specialized interpreters) have sometimes proven to outperform handwritten compilers, e.g., for Action Semantics [15].

In action semantics, one (1) expands a source program into the corresponding action, (2) analyzes and transforms this action, and then (3) translates it into a target program:



(1) Expanding a source program into an action is usually performed by dedicated translators (ASD tools, etc.). It can, however, also be achieved using the first Futamura projection, by specializing an interpreter for the source language written in action notation.

(2) The dashed arrow could be directly obtained using the first Futamura projection, by specializing an interpreter for the source language written in the target language. It is our experience that this dashed arrow is more efficient in practice than composing (1), (2) and (3), especially in the absence of (2). In effect, the dashed arrow "deforests" an intermediate result. However, (2) is one of the *raisons d'être* of action semantics, which offers an algebra for analyzing and transforming actions [15].

(3) The target language of existing actions compilers is C [5] or assembly language [16, 18]. Compiling action notation can also be achieved using the first Futamura projection, as described below.

Bondorf and Palsberg compile action notation by specializing an action interpreter written in Scheme [3, 4]. Their result is comparable in efficiency to the results of other systems [5, 16], both in terms of compile time and in terms of run time.

Bondorf and Palsberg's result is significant in two ways:

- extensionally: Along with Jørgensen's Futamura projections of an interpreter for a lazy functional language [13], it is one of the most successful applications of the Similix partial evaluator.
- intensionally: Compiling action notation is a non-trivial affair, which required modifying both Similix and the action interpreter. The action interpreter was subjected to extensive "binding-time improvements" (i.e., local program transformations making a partial evaluator yield better results; Bondorf and Palsberg dedicate a third of their article to them), and a second pass of partial evaluation on one residual program was observed to yield another speedup factor of 2.

This work: Instead of using Similix, we compile action notation using a type-directed partial evaluator [7]. Type-directed partial evaluation is:

- simpler: no binding-time improvements are necessary, and the result is obtained in one pass;
- at least as effective: the output is perceptibly more efficient than Similix's output; and
- more efficient: type-directed partial evaluation appears to use about 7 times less space and to be about 28 times faster than Similix.

This article: Section 2 compares and contrasts Similix and our type-directed partial evaluator. Section 3 revisits Bondorf and Palsberg's benchmarks. Section 4 reports further benchmarks. Section 5 concludes.

2 Syntax-Directed vs. Type-Directed Partial Evaluation

2.1 Syntax-directed partial evaluation (sdpe)

The Similix partial evaluator is offline, syntax-directed, polyvariant, and self-applicable [1]. In general, it is not idempotent.

- Offline: Similix performs a number of analyses over the source program to optimize specialization (and enable self-application). These analyses include a binding-time analysis.

- Syntax-directed: Similix operates on the text of the source program. As pointed out below, the corresponding interpretive overhead is eliminated by self-application. Similix manages its own environment and represents values symbolically. For example, it represents higher-order functions as closures.

- Polyvariant: some source program points are selected as "specialization points." They give rise to several (mutually recursive) specialized instances in the residual program.

- Self-applicable: rather than running Similix on a source program,[1] one can first specialize Similix with respect to this source program,[2] and use the resulting dedicated specializer. As illustrated here, the result of the second Futamura projection is about 3 times more efficient than the first Futamura projection, both in time and space.

- Not idempotent: a second pass of partial evaluation on the binding-time improved action interpreter yields programs that are 3.0 times faster and use 9.9 times less space than the output of the first pass. Iterating this experiment, we have observed that a stable residual program is obtained after the third pass. Subsequent passes usually yield essentially the same programs as the third pass, modulo renaming, the order of parameters in residual procedures, and the occasional trivial let-binding — and sometimes also modulo unfolding one more iteration of a residual loop per extra pass. Programs beyond the third pass appear to have the same time and space behaviour. These observations also hold for the original, unimproved action interpreter.

[1] When the source program is an interpreter, this instance of partial evaluation is known as "the first Futamura projection."

[2] When the source program is an interpreter, this instance of self-application is known as "the second Futamura projection."

In addition, Similix includes a post-processing phase, which performs residual constant folding and unfolds residual functions that are called only once. This post-processing phase essentially performs online monovariant partial evaluation.

2.2 Type-directed partial evaluation (tdpe)

The partial evaluator we use here is online, type-directed, monovariant, self-applicable, and idempotent [7, 8].

- Online: primitive operators probe their operands to decide whether to proceed with a static computation or residualize the operation.
- Type-directed: the source program is already compiled, and its type is the only information supplied to the partial evaluator. The partial evaluator relies on the underlying representations of values. In particular, it does not use an environment.
- Monovariant: our type-directed partial evaluator only unfolds or residualize calls — it does not "clone" them. To use the partial-evaluation jargon, there is no polyvariant program-point specialization [6, 11].
- Self-applicable: the partial evaluator can be specialized with respect to a type. The resulting improvement is insignificant.
- Idempotent: a second pass of partial evaluation yields programs that are textually identical to the output of the first pass.

There is no post-processing phase.

2.3 Comparing the input to syntax-directed partial evaluation and the input to type-directed partial evaluation

2.3.1 Similarities

The input to Similix and the input to type-directed partial evaluation are essentially the same action interpreter written in Scheme.

2.3.2 Differences

Similix expects the name of a function to specialize, the static arguments with respect to which this function should be specialized, and a textual representation of the action interpreter. Type-directed partial evaluation expects a higher-order value and a representation of its type.

The input to Similix uses Scheme's underlying recursion, whereas the input to type-directed partial evaluation uses a fixed-point operator to implement dynamic loops. Similarly, the input to Similix uses Scheme's underlying conditional expressions, whereas the input to type-directed partial evaluation uses a function expecting a boolean and two thunks to implement tests.

2.3.3 Lambda count

There are one dynamic loop and five dynamic tests in each action interpreter. The following table displays the number of lambda-abstractions in each interpreter.

action interpreter	lambda-abstractions
original for sdpe	43
improved for sdpe	43
raw for tdpe	78
cooked for tdpe	48

In the "raw" row, we use higher-order functions for conditional expressions, recursive definitions, etc. We have tried to compensate for the higher-order overhead by macro-expanding these forms into native conditional expressions, recursive definitions, etc. The resulting numbers are displayed in the "cooked" row.

2.4 Comparing the output of syntax-directed partial evaluation and the output of type-directed partial evaluation

2.4.1 Similarities

The output of Similix (both after one pass and after a second pass) and the output of type-directed partial evaluation share the same overall structure, which is inherited from the source program.

2.4.2 Differences

Type-directed partial evaluation outputs one lambda-abstraction, whereas Similix outputs several (mutually recursive) definitions. Similix is polyvariant and thus the same function may be called from several places in the residual program, possibly recursively. At specialization points, Similix uses static projections to extract the dynamic components of higher-order values and partially static structures; because of this extraction, the arity of residual definitions differs from the arity of source definitions. To obtain a similar effect using type-directed partial evaluation, the user would need to include such projections explicitly, in a way comparable to “type specialization” [9].

2.4.3 Lambda count

The following table displays the number of lambda-abstractions in the residual programs considered in this article. As a consequence of the different ways of implementing dynamic tests and dynamic loops, the output of type-directed partial evaluation contains many more lambda-abstractions than the output of Similix.

output from	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe (1st pass)	104	142	107	63
sdpe (2nd pass)	23	30	22	16
sdpe (3rd pass)	22	30	26	14
sdpe (4th pass)	22	32	34	15
tdpe (raw)	273	329	259	168
tdpe (cooked)	171	239	216	131

3 Benchmarks and Assessments

Bondorf and Palsberg’s experiments took place on a SPARC 1 using SCM. We reproduced these experiments on a Silicon Graphics Iris4d running IRIX 6.3, using Chez Scheme version 5.0, Similix 5.0 [2], and an online type-directed partial evaluator [8]. Besides the definitional action interpreter written in Scheme, the experiments involved two interpreters written in action notation: one for Lee’s HypoPL [14] and one for a “substantial” subset of Ada [16]. The four test programs were a bubblesort program (both in HypoPL and in Ada), Eratosthenes’s sieve (in Ada), and Euclid’s algorithm (in Ada).

As illustrated by the diagram of Section 1, we first derived an action out of one of the two interpreters and one of the four test programs, and then we specialized the action interpreter with respect to this test action. For Similix, we used the binding-time improved action interpreter, since it yields the best results. For type-directed partial evaluation, we used the original action interpreter. (Section 4.1 describes the results of specializing the original interpreter with Similix and of specializing the binding-time improved interpreter with type-directed partial evaluation.)

We ran three series of experiments: two with Similix and one with type-directed partial evaluation.

1. First Futamura projection with Similix: we directly specialize the action interpreter with respect to each of the test actions.
2. Result of the second Futamura projection with Similix: we use the second Futamura projection to obtain a specializer dedicated to the action interpreter, and we then (indirectly) specialize this interpreter with respect to each of the test actions.
3. Type-directed partial evaluation: we directly specialize the action interpreter with respect to each of the test actions.

Each of the numbers below is the average of 10 runs. We should stress that a garbage collector such as Chez Scheme's makes it difficult to benchmark the timings of a computation — we have zero control on the layouts of the heap and of the code and their incidence on the processor cache(s). For example, in Sections 3.1 and 3.2, the timings of Similix's post-processing phase differ by about 9%, even though the computations are the same and yield identical results. On the other hand, the space measures are identical. To reduce environmental hazards, we have conducted all these measures off hours and on a little-loaded machine.

3.1 First Futamura projection with Similix

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	645	668	673	664
Process	2 518	2 975	2 266	1 391
Postprocess	582	751	547	286
Total	3 745	4 394	3 486	2 341

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	1.03	1.03	1.03	1.03
Process	8.39	9.89	7.74	4.48
Postprocess	1.08	1.42	0.91	0.52
Total	10.50	12.34	9.68	6.03

3.2 Result of the second Futamura projection with Similix

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	516	624	569	305
Postprocess	581	819	542	303
Total	1 097	1 443	1 111	608

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	2.33	2.90	2.37	1.21
Postprocess	1.08	1.42	0.91	0.52
Total	3.41	4.32	3.28	1.73

3.3 Type-directed partial evaluation

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	41	50	38	21

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	0.49	0.59	0.44	0.27

3.4 Assessment of the compilers

- On the average, using the result of second Futamura projection yields results that are *3.4 times more efficient in time* and *3.1 times more efficient in space* than the first Futamura projection.
- Discounting preprocessing, using the result of the second Futamura projection yields results that are *2.7 times more efficient in time and space* than the first Futamura projection, on the average.
- On the average, type-directed partial evaluation is *28.5 times more efficient in time* and *7.0 times more efficient in space* than the result of Similix's second Futamura projection.

3.5 Assessment of the residual programs

The residual programs are textually identical for the two instances of Similix, and have about the same size as the result of type-directed partial evaluation.

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	51	224	150	50
raw tdpe	63	197	175	90
cooked tdpe	17	182	80	10

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	0.70	1.31	2.03	0.67
raw tdpe	1.03	1.92	2.27	1.26
cooked tdpe	0.09	0.42	0.42	0.15

- On the average, raw type-directed partial evaluation yields programs that are *1.3 times less efficient in time* and *1.5 times less efficient in space* than the output of Similix.

- On the average, cooked type-directed partial evaluation yields programs that are *2.8 times more efficient in time* and *5.0 times more efficient in space* than the output of Similix.

3.6 Assessment of the interpreters

Since we have just compared the performances of the specialized versions of the action interpreter, it seems natural to report the performances of these interpreters. Here are the measures, both for the original interpreter and the binding-time improved interpreter (for syntax-directed partial evaluation), and both for the raw and the cooked versions of the original interpreter.

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
original for sdpe	8 001	11 864	16 865	6 820
improved for sdpe	8 307	11 832	14 925	6 850
raw for tdpe	8 199	11 531	14 090	6 170
cooked for tdpe	7 440	11 432	14 250	5 260

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
original for sdpe	29.72	41.42	51.77	23.70
improved for sdpe	30.73	42.87	53.51	24.81
raw for tdpe	31.72	43.81	54.82	25.21
cooked for tdpe	29.06	40.41	50.33	22.90

- On the average, the binding-time improvements tax the action interpreter by about 2% in time and 4% in space.

- On the average, cooking improves the action interpreter by about 7% in time and 9% in space.

- On the average, the [cooked] original interpreter for type-directed partial evaluation is marginally more efficient than the original interpreter for syntax-directed partial evaluation (15% in time and 3% in space).

The comparisons between the source interpreters and the residual programs do not match: the source interpreters are essentially comparable in efficiency (15% in time and 3% in space), whereas the residual programs are not comparable (280% in time and 500% in space).

To conclude this section, let us indulge in the traditional effectiveness measure of partial evaluation [11]:

- On the average, syntax-directed partial evaluation [of the improved interpreter] yields a program that runs 115 times faster and consumes 34 times less space than running the original interpreter.
- On the average, type-directed partial evaluation yields a program that runs 301 times faster and consumes 173 times less space than running the original interpreter.

4 Further Benchmarks and Assessments

This section briefly describes some of the most significant other benchmarks we conducted [17, Chapter 5].

4.1 On Similix's binding-time improvements

Bondorf and Palsberg need to improve the binding times of the action interpreter to specialize it better. The binding-time improvements exert a cost during specialization, but they yield better residual programs.

- On the average, the first Futamura projection over the binding-time improved action interpreter is about 1.2 times slower than over the original, unimproved interpreter, and consumes about 1.4 times more space.
- On the average, the result of the second Futamura projection over the binding-time improved action interpreter is about 1.8 times slower than over the original interpreter, and consumes about 1.4 times more space.
- As for the residual programs, they are 1.6 times smaller, about 1.9 times faster, and consume about 1.8 times less space, on the average.

We have also specialized the binding-time improved action interpreter using type-directed partial evaluation. On the average, specialization consumes about 2% more time and 6% more space. The residual programs only differ because some variables bound to the empty tuple are replaced by a call to the primitive function constructing an empty tuple. They have thus essentially the same size and they use the same space and the same time.

4.2 On Similix's polyvariance

Similix performs polyvariant program-point specialization, cloning selected source specialization points in residual programs. The action interpreter contains 5 such specialization points. Each of the source programs gives rise to the following number of residual specialized points. Similix then post-unfolds those called only once.

Number of points	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Before post-unfolding:	176	212	166	111
After post-unfolding:	12	16	12	6

We have not been able to measure the cost of polyvariance in Similix's specializer: each access to its working list is too atomic.

In any case, regardless of polyvariance, we observe that on the average, Similix's post-processing phase takes 5.7 times more time and 1.5 times more space than type-directed partial evaluation.

4.3 On Similix's post-unfolding phase

Users can switch Similix's post-unfolding (but not its post-constant folding) on or off. On the average, switching post-unfolding off speeds up post-processing by about 10% and makes it consume about 20% more space. It yields residual programs that are 3.4 times bigger, run about 1.5 times slower, and consume 1.4 times more space, on the average.

These measures hold both for the first and for the second Futamura projections.

4.4 On Similix's second pass

Bondorf and Palsberg mention that a second pass of partial evaluation on one of their residual programs made it twice as fast, for a cost of ten times the overall compile time. Here are the complete numbers.

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	829	1 125	877	452
Process	344	6 681	2 434	179
Postprocess	391	174 476	42 482	95
Total	1 564	182 282	45 793	726

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	1.11	1.49	1.17	0.63
Process	0.82	9.60	5.21	0.37
Postprocess	0.93	840.70	186.53	0.16
Total	2.86	851.79	192.91	1.16

To terminate, the second pass requires one to hand-annotate the residual program with a generalization directive. (The action interpreter is annotated with a similar directive.)

On the average, but with much variance, the second pass is about 50 times slower than the result of the second Futamura projection and it uses 82 times as much space.

The second pass inlines several functions and unfolds one more loop. Here are the sizes of the residual programs, counting the number of cons-cells of their Scheme representation:

Size (pairs)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe (first pass)	2 803	3 463	2 662	1 599
sdpe (second pass)	2 183	15 389	8 579	999
tdpe	2 795	3 295	2 508	1 668

These numbers illustrate the tradeoff between unfolding and reducing during the second pass.

The effect of Similix's second pass is thus global, in that the code is restructured, and local, in that Similix performs additional peephole constant optimizations (e.g., conditional expressions with a literal test). Here are the numbers of program points before and after the second pass:

Number of points	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
After the first pass:	12	16	12	6
After the second pass:	8	9	7	4

We have observed that the number of residual recursive definitions matches the number of residual fixed-point operations in the output of type-directed partial evaluation.

On the average, the second pass yields residual programs that are 3.00 times faster and use 9.92 times less memory than the output of the first pass. Here are the detailed performances of the residual programs, both after a second pass of syntax-directed partial evaluation and after one pass of type-directed partial evaluation. The raw and the cooked rows were described in Section 3.5.

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	15	115	90	10
raw tdpe	63	197	175	90
cooked tdpe	17	182	80	10

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	0.05	0.18	0.23	0.07
raw tdpe	1.03	1.92	2.27	1.26
cooked tdpe	0.09	0.42	0.42	0.15

- On the average, raw type-directed partial evaluation yields programs that are *4.21 times less efficient in time* and *14.78 times less efficient in space* than the output of the second pass of Similix.

- On the average, cooked type-directed partial evaluation yields programs that are *1.15 times less efficient in time* and *2.03 times less efficient in space* than the output of the second pass of Similix.

4.5 On Similix's idempotence

As mentioned in Section 2, it is our observation that Similix is essentially idempotent after three passes for the binding-time improved action interpreter. As for type-directed partial evaluation, it is idempotent after one pass. For the record, here are the time and space performances of Similix's third and fourth passes, and of type-directed partial evaluation's second pass.

4.5.1 Syntax-directed partial evaluation (third pass)

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	553	18 048	5 887	271
Process	159	3 583	1 577	119
Postprocess	234	41 074	11 242	99
Total	946	62 705	18 706	489

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	0.63	0.67	0.35	0.31
Process	0.32	0.53	0.27	0.18
Postprocess	0.52	16.08	4.59	0.21
Total	1.47	17.28	5.21	0.70

4.5.2 Syntax-directed partial evaluation (fourth pass)

Time ms	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	610	19 163	6 706	486
Process	267	4 077	1 924	324
Postprocess	273	41 963	11 886	196
Total	1 150	65 203	20 516	1 006

Space Mb	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	0.63	6.97	3.91	0.39
Process	0.31	4.56	2.92	0.23
Postprocess	0.53	161.54	46.34	0.24
Total	1.47	173.07	53.17	0.86

4.5.3 Type-directed partial evaluation (second pass)

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	13	18	13	9

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	0.19	0.22	0.17	0.11

4.5.4 Assessment

The numbers in Sections 4.5.2 and 4.5.3 describe the overhead of a "no-op," since source and residual programs are the same (stable for Section 4.5.2 and textually identical for Section 4.5.3). Let us take these numbers as indicative of the *standard overhead of partial evaluation for the action interpreter*. Subtracting this overhead from the earlier numbers accounting for the original partial evaluation gives us a measure of the actual static computation performed by each partial evaluator.

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe (1st pass)	-53	-63 760	-19 405	-398
sdpe (2nd pass)	414	117 079	25 277	-280
sdpe (3rd pass)	-204	-2 498	-1 810	-517
tdpe	28	32	25	12

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe (1st pass)	1.94	-168.75	-49.89	0.87
sdpe (2nd pass)	1.39	678.72	139.74	0.30
sdpe (3rd pass)	0.00	-155.79	-47.96	-0.16
tdpe	0.30	0.37	0.27	0.16

Negative numbers indicate that Similix sometimes uses less than its standard overhead.

According to these measures, type-directed partial evaluation performs static computations in a way that is more economical than syntax-directed partial evaluation.

4.6 On Similix's binding-time improvements, revisited

Repeatedly specializing the original (unimproved) action interpreter yields a stable residual program essentially as quickly as specializing the binding-time improved one. Three passes are sufficient to yield a fixed point for the HypoPL bubble-sort program. For the Ada bubble-sort program, however, Similix unfolds one iteration of a loop at each step. As for the two other Ada programs, they yield a stable program after four iterations. This experiment was quite time consuming,[3] but it shows that binding-time improvements, by making source programs specialize better, actually do not contribute to accelerating the convergence of repeated partial evaluation. At any rate, binding-time improvements are not monotonic: for example, their complete opposite, which amounts to classifying every component of the source program as dynamic, also yields a stable (and trivial) residual program, in one pass (modulo post-processing).

[3] Iterating partial evaluation with Similix is not completely trivial since one needs to hand-annotate each of the successive specialized programs to ensure that the next pass terminates.

5 Conclusion and Issues

Our work stems from comparing syntax-directed and type-directed partial evaluation [17, Chapter 5]. We have taken what has been presented as a significant application of an offline syntax-directed partial evaluator [3, 4], we have reproduced it, we have explored it further, and we have compared it to using an online type-directed partial evaluator.

5.1 Executive summary

The following comparative tables summarize our results. They display the proportion of time and space consumed, relative to the best instance of using type-directed partial evaluation.

5.1.1 Compiling source actions

	Time	Space
1st FP, Similix (original)	71.49	15.41
2nd FP, Similix (original)	23.09	5.71
1st FP, Similix (improved)	95.61	21.67
2nd FP, Similix (improved)	28.45	7.04
Similix 2nd pass (improved)	1230.86	473.07
tdpe	1.00	1.00

5.1.2 Running target programs

	Time	Space
from Similix (original)	5.32	9.00
from Similix (improved)	2.80	5.00
from Similix 2nd pass	0.87	0.49
from tdpe (raw)	3.66	7.28
from tdpe (cooked)	1.00	1.00

5.1.3 Running interpreters

	Time	Space
for Similix (original)	1.15	1.03
for Similix (improved)	1.17	1.07
for tdpe (raw)	1.07	1.09
for tdpe (cooked)	1.00	1.00

5.1.4 Effect of Similix

	Time	Space
running interpreter (original)	114.67	33.74
running interpreter (improved)	116.96	35.09
running target program (original)	1.90	1.80
running target program (improved)	1.00	1.00

5.1.5 Effect of type-directed partial evaluation

	Time	Space
running interpreter (raw)	322.23	188.47
running interpreter (cooked)	301.15	172.91
running target program (raw)	3.99	7.46
running target program (cooked)	1.00	1.00

5.2 Syntax-directed vs. type-directed partial evaluation

Our conclusions are that:

- the full power of offline partial evaluation (binding-time analysis, binding-time improvements, and polyvariant specialization) is not needed for compiling action notation; and
- for compiling action notation, the overhead of Similix is significant compared to that of type-directed partial evaluation, especially if one takes into account further passes.

5.3 Compile-time vs. run-time performances

On the other hand, we are only concerned with compile times: Bondorf and Palsberg's results are competitive with other compilers for action notation, and we improve on these results by about one order of magnitude in time. The partial-evaluation technology we are using is otherwise the same, and the resulting compiled code is thus essentially the same. Such Scheme code is considerably less efficient than, e.g., the assembly-language output of Ørbæk's optimizing action-based compiler generator Oasis [18]. On the other hand, Oasis's compile times are considerably larger than those of type-directed partial evaluation.

5.4 Partial-evaluation optimality and multi-level specialization

Optimality is an elusive concept in partial evaluation [11]. For the purpose of this conclusion, let us take idempotence as an optimality criterion: partial evaluation is optimal if one more pass yields the same residual program. This criterion is compatible with the usual concept of normalization and normal forms, e.g., in the lambda-calculus.

According to this criterion, the fact that Similix is not idempotent indicates that it is sub-optimal: its binding-time analysis is too conservative. This conservativeness poses a challenge for extending Similix into a multi-level partial evaluator [10].

The idea of multi-level partial evaluation is that static information becomes available in stages, and that correspondingly, specialization should also occur in stages. The offline approach to multi-level partial evaluation generalizes the offline approach to partial evaluation, where static information becomes available in one stage. As such, it assumes that a multi-level binding-time analysis can predict enough of the successive static data flows to improve the corresponding successive specializations, *without having to perform successive binding-time analyses* as the successive static data become available. In that sense, a multi-level binding-time analysis factors the successive single-level binding-time analyses out of the corresponding stages of offline partial evaluation. However, our experiment demonstrates that a single-level binding-time analysis can miss a significant portion of the static data flow. It seems likely that these misses are compounded in a multi-level binding-time analysis, and thus that an incremental strategy using online type-directed partial evaluation constitutes a viable alternative solution to offline multi-level partial evaluation.

Acknowledgements

Thanks to Anders Bondorf and Jens Palsberg for letting us access the code of their action interpreter. We are also grateful to Karoline Malmkjær, Peter Mosses, and Peter Sestoft for encouraging us to report these results and for commenting on an earlier version of this article. Finally, we would like to thank Oscar Waddell for sharing with us his experience in making benchmarks in the presence of garbage collection.

The core of this work was carried out at the University of Aarhus, Denmark in the fall of 1997. This article was completed during a visit to Hanyang University, Korea by the second author, in the spring of 1998, supported by the Korean Science and Engineering Foundation under grant 971-0903-026-2. We are grateful to Prof. Kyung-Goo Doh for his invitation and his support.

A Benchmarks with the original, unimproved interpreter

A.1 First Futamura projection with Similix

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	631	604	626	610
Process	1 803	1 933	1 387	925
Postprocess	530	573	424	299
Total	2 964	3 110	2 437	1 834

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Preprocess	0.98	0.98	0.98	0.98
Process	5.58	6.30	4.76	2.86
Postprocess	1.06	1.22	0.89	0.65
Total	7.62	8.50	6.63	4.49

A.2 Result of the second Futamura projection with Similix

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	429	474	372	245
Postprocess	534	602	441	300
Total	963	1 076	813	545

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
Process	1.77	2.06	1.67	0.89
Postprocess	1.06	1.22	0.90	0.64
Total	2.83	3.28	2.57	1.53

A.3 Performance of the residual programs

Time (ms)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	129	286	275	100
tdpe (raw)	53	205	155	50
tdpe (cooked)	17	182	80	10

Space (Mb)	`bubble.hpl`	`bubble.ad`	`sieve.ad`	`euclid.ad`
sdpe	1.60	2.23	3.11	1.11
tdpe (raw)	1.03	1.92	2.30	1.30
tdpe (cooked)	0.09	0.42	0.42	0.15

References

[1] Anders Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1990. DIKU Rapport 90/17.

[2] Anders Bondorf. Similix 5.0 manual. Technical report, DIKU, Computer Science Department, University of Copenhagen, Copenhagen, Denmark, 1993. Included in the Similix 5.0 distribution.

[3] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In Arvind, editor, *Proceedings of the Sixth ACM Conference on Functional Programming and Computer Architecture*, pages 308–317, Copenhagen, Denmark, June 1993. ACM Press.

[4] Anders Bondorf and Jens Palsberg. Generating action compilers by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[5] Deryck F. Brown, Hermano Moura, and David A. Watt. Actress: an action semantics directed compiler generator. In Uwe Kastens and Peter Pfahler, editors, *Proceedings of CC'92, the 4th International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 95–109, Paderborn, Germany, October 1992. Springer-Verlag.

[6] 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.

[7] 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.

[8] 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 BRICS RS-97-53.

[9] Olivier Danvy. A simple solution to type specialization. Technical Report BRICS RS-98-1, Department of Computer Science, University of Aarhus, Aarhus, Denmark, January 1998. Extended version to appear in the proceedings of ICALP'98.

[10] Robert Glück and Jesper Jørgensen. Fast binding-time analysis for multi-level specialization. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Perspectives of System Informatics. Proceedings*, volume 1181 of *Lecture Notes in Computer Science*, pages 261–272. Springer-Verlag, 1996.

[11] 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.

[12] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. MIX: A self-applicable partial evaluator for experiments in compiler generation. *LISP and Symbolic Computation*, 2(1):9–50, 1989.

[13] Jesper Jørgensen. Generating a compiler for a lazy language by partial evaluation. In Andrew W. Appel, editor, *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, pages 258–268, Albuquerque, New Mexico, January 1992. ACM Press.

[14] Peter Lee. *Realistic Compiler Generation.* The MIT Press, 1989.

[15] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[16] Jens Palsberg. *Provably Correct Compiler Generation.* PhD thesis, Department of Computer Science, University of Aarhus, Aarhus, Denmark, 1992.

[17] Morten Rhiger. A study in higher-order programming languages. Master's thesis, DAIMI, Department of Computer Science, University of Aarhus, Aarhus, Denmark, December 1997.

[18] Peter Ørbæk. OASIS: An optimizing Action-based compiler generator. In Peter A. Fritzson, editor, *Proceedings of CC'94, the 5th International Conference on Compiler Construction*, number 786 in Lecture Notes in Computer Science, pages 1–15, Edinburgh, U.K., April 1994. Springer-Verlag.

Recent BRICS Report Series Publications

RS-98-13 **Olivier Danvy and Morten Rhiger.** ***Compiling Actions by Partial Evaluation, Revisited*****. June 1998. 25 pp.**

RS-98-12 **Olivier Danvy.** ***Functional Unparsing*****. May 1998. 7 pp. This report supersedes the earlier report BRICS RS-98-5. Extended version of an article to appear in** ***Journal of Functional Programming*****.**

RS-98-11 **Gudmund Skovbjerg Frandsen, Johan P. Hansen, and Peter Bro Miltersen.** ***Lower Bounds for Dynamic Algebraic Problems*****. May 1998. 30 pp.**

RS-98-10 **Jakob Pagter and Theis Rauhe.** ***Optimal Time-Space Trade-Offs for Sorting*****. May 1998. 12 pp.**

RS-98-9 **Zhe Yang.** ***Encoding Types in ML-like Languages (Preliminary Version)*****. April 1998. 32 pp.**

RS-98-8 **P. S. Thiagarajan and Jesper G. Henriksen.** ***Distributed Versions of Linear Time Temporal Logic: A Trace Perspective*****. April 1998. 49 pp. To appear in** ***3rd Advanced Course on Petri Nets*****, ACPN '96 Proceedings, LNCS, 1998.**

RS-98-7 **Stephen Alstrup, Thore Husfeldt, and Theis Rauhe.** ***Marked Ancestor Problems (Preliminary Version)*****. April 1998. 36 pp.**

RS-98-6 **Kim Sunesen.** ***Further Results on Partial Order Equivalences on Infinite Systems*****. March 1998. 48 pp.**

RS-98-5 **Olivier Danvy.** ***Formatting Strings in ML*****. March 1998. 3 pp. This report is superseded by the later report BRICS RS-98-12.**

RS-98-4 **Mogens Nielsen and Thomas S. Hune.** ***Deciding Timed Bisimulation through Open Maps*****. February 1998.**

RS-98-3 **Christian N. S. Pedersen, Rune B. Lyngsø, and Jotun Hein.** ***Comparison of Coding DNA*****. January 1998. 20 pp. To appear in** ***Combinatorial Pattern Matching: 9th Annual Symposium*****, CPM '98 Proceedings, LNCS, 1998.**

RS-98-2 **Olivier Danvy.** ***An Extensional Characterization of Lambda-Lifting and Lambda-Dropping*****. January 1998.**

RS-98-1 **Olivier Danvy.** ***A Simple Solution to Type Specialization (Extended Abstract)*****. January 1998. 7 pp.**