Interpretive Overhead and Optimal Specialisation. Or: Life without the Pending List (Workshop Version)

Lars Hartmann, Neil D. Jones, Jakob Grue Simonsen

DIKU (Computer Science Dept., University of Copenhagen, Denmark)

Abstract. A self-interpreter and a program specialiser with the following characteristics are developed for a simple imperative language:
1) The self-interpreter runs with *program-independent interpretive overhead*; 2) the specialiser achieves *optimal specialisation*, that is, it eliminates *all interpretation overhead*; 3) the specialiser has been run on a variety of small and large programs, including specialising the self-interpreter to itself; 4) all specialiser parts except for loop unfolding have been proven to terminate.
We achieve the above by using a structured language with *separated control and data flow*, containing loops but without `while`. The specialiser uses two-level binding-time annotations in a new way: source annotations are *used to ensure correctness* of specialised programs. A novelty: the specialiser *has no need for a pending list,* and does *no call graph analysis* of the residual program. A source-to-source *normalisation phase* does program transformations to avoid situations where the specialiser would need to specialise code based on an unknown state. A *pruning phase* efficiently achieves the effect of Romanenko's arity raising.

Two interesting lines of work concern self-interpreters for programming languages. One line is to develop a self-interpreter with *program-independent interpretation overhead*; this was the basis for the linear-time complexity hierarchy of [27,25,14,16,24,4]. Another line is to develop a program specialiser[1] and a self-interpreter that allow *optimal program specialisation*, a measure of the strength/quality of a program specialiser, discussed in [18] Sections 6.4 and 8.5.1, and in [9,8,10,19,28][2].

The technical breakthrough for each line was to construct a self-interpreter with a certain property. We know of no prior self interpreter `sint` that simultaneously possesses *both properties*:

- `sint` runs programs with program-independent overhead; and
- `sint` can be specialised optimally.

[1] Also known as a *partial evaluator*.

[2] As known from many fields, "optimality" is a very slippery concept. The formulation for the strength/quality of a specialiser in [18] turned out to be practically useful, and has since been dubbed Jones-optimality.

The above was our initial goal. Bottom line: the goal was achieved (see [11] for details) and, along the way, new insights were gained into the control structure of specialised programs; relations between binding-time annotations, conditionals and loop unfolding rules; the use of program transformation to allow more liberal unfolding; and how to specialise programs with tree-structured values.

1 Programming languages, interpretation overhead and optimality

Basics. Program specialisation, or partial evaluation, is a well-established automatic program transformation [18,6]. Its purpose is to speed up programs by exploiting partial, compile-time, knowledge of the subject program's run-time input. Standard jargon is to use the term *static* for that part of the program's input known at specialisation time, and *dynamic* for that part of the input only known at run-time. We briefly recapitulate central notions concerning interpreters and specialisers; for an in-depth coverage, the reader is referred to [18,16].

A *programming language* $\mathbf{L}$ consists of a set of programs $\mathbf{Prog}$, a set of data $\mathbf{D}$, and a semantic function $[\![\_]\!] : \mathbf{Prog} \to (\mathbf{D} \rightharpoonup \mathbf{D})$ that assigns to each $\mathtt{p} \in \mathbf{Prog}$ a partial input-output function $[\![\mathtt{p}]\!] : \mathbf{D} \rightharpoonup \mathbf{D}$. A *timed programming language* has in addition a function $time_p(\mathtt{d})$ assigning a running time (a positive integer) to each input, such that $[\![\mathtt{p}]\!](\mathtt{d})$ is defined if and only if $time_p(\mathtt{d})$ is defined.

We further assume that the data set is *closed under pairing*, meaning $\mathbf{D} \times \mathbf{D} \subseteq \mathbf{D}$; and that the language has *concrete syntax*, meaning $\mathbf{Prog} \subseteq \mathbf{D}$. Write $\mathtt{d}_1 \doteq \mathtt{d}_2$ to indicate partial equality: that both sides are undefined, or both sides are defined and equal.

Interpretation and its overhead. A *self-interpreter* is a program `sint` that satisfies

$$\forall \mathtt{p} \forall \mathtt{d}([\![\mathtt{sint}]\!](\mathtt{p},\mathtt{d}) \doteq [\![\mathtt{p}]\!](\mathtt{d}))$$

The *interpretation overhead* can be measured by the ratio

$$overhead_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) = time_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) / time_{\mathtt{p}}(\mathtt{d})$$

In general practice, a self-interpreter will satisfy

$$\forall \mathtt{p} \exists c \forall \mathtt{d}(\ overhead_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) \leq c\ )$$

that is, interpreting a program incurs (at most) a slowdown of a factor of c in relation to running the program natively. Factor c can depend on `p`, so the overhead is *program-dependent* in general. Typical causes of program-dependent overhead can be the need for the interpreter to look up variables in a run-time store, or to find the target of a function call or a `goto` command.

We say that the self-interpreter has **program-independent** interpretation overhead if the overhead does not depend on the program p, that is, the following holds (note the changed order of the quantifiers):

$$\exists c \forall \mathtt{p} \forall \mathtt{d}(\ \mathit{overhead}_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) \leq c\ )$$

Such an interpreter is called "efficient" (see e.g. [16, Def. 19.1.1]). There exist efficient self-interpreters, e.g., for structured programs with only one variable. This has been proven in [14,16,2,24] where it is shown to lead to a complexity-theoretic *linear hierarchy* theorem: that even within linear time, for such a language, increasing an allowed running time bound properly increases the class of decision problems that can be solved within the given bound.

Specialisation and optimality. A *program specialiser* is a program spec that satisfies

$$\forall \mathtt{p} \forall \mathtt{s} \forall \mathtt{d}([\![[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})]\!](\mathtt{d}) \doteq [\![\mathtt{p}]\!](\mathtt{s},\mathtt{d}))$$

We call $[\![\mathtt{spec}]\!](\mathtt{p},\mathtt{s})$ the *specialised program*, and denote it for short by $\mathtt{p}_\mathtt{s}$. In general practice, the result of program specialisation will satisfy :

$$\forall \mathtt{p} \exists c' \forall \mathtt{s} \forall \mathtt{d}(\ \mathit{speedup}_{\mathtt{p}}(\mathtt{s},\mathtt{d}) \geq c'\ ) \text{ where } \mathit{speedup}_{\mathtt{p}}(\mathtt{s},\mathtt{d}) = \frac{\mathit{time}_{\mathtt{p}}(\mathtt{s},\mathtt{d})}{\mathit{time}_{\mathtt{p}_\mathtt{s}}(\mathtt{d})}$$

The point of specialisation is speedup: $\mathtt{p}_\mathtt{s}$ may be substantially faster than p.

The optimality criterion arose as a precise criterion for being able to state that a partial evaluator is "good enough", that is, able to remove as much static overhead as can reasonably be expected. This criterion is rather vague, but as specialisers have been much applied to program interpreters as well-known examples of programs with significantly large static overheads, the following more precise—and ambitious—criterion can be stated: Given a self-interpreter sint, the specialiser should ideally be able *to remove all interpretation overhead*.

Technically, this can be formulated as follows: given a program p, construct program $\mathtt{p}' = [\![\mathtt{spec}]\!](\mathtt{sint},\mathtt{p})$ by specialisation. It is straightforward to see that this transformed program is semantically equivalent to p, i.e., that $[\![\mathtt{p}']\!] = [\![\mathtt{p}]\!]$. The specialiser is called optimal (Definition 6.4 of [18]) if $\mathtt{p}'$ is always at least as fast as p, that is, for all input data d we have

$$\mathit{time}_{\mathtt{p}}(\mathtt{d}) \geq \mathit{time}_{\mathtt{p}'}(\mathtt{d})$$

Another way to say this is:

$$\mathit{speedup}_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) \geq \mathit{overhead}_{\mathtt{sint}}(\mathtt{p},\mathtt{d})$$

In words: $\mathtt{p}'$ suffers from none of the overhead that was introduced by use of the interpreter sint.

This goal, while often stated, is not often achieved. Remark: optimality is more a property of *the strength of the specialiser* than of the self-interpreter.

2 Designing LOOP

Our two goals: program-independent interpretation overhead and optimal specialisation. The program-independent requirement on interpretation overhead is quite stringent: It rules out the use of an unbounded number of program variables as it would take program-dependent time to search the store or value environment; and it also rules out the use of unstructured control such as `goto` or calls to named functions, as these would also need program-dependent searches in the program symbol table. One step to circumventing these difficulties is to use an imperative language with structured control. Another step is to follow the LISP/Scheme/ML tradition of tree-structured data, and to limit programs to have at most a single variable with only one leaf value `N` (nil). Perhaps surprisingly this does not cause any loss of expressiveness in the sense that all Turing-computable function may still be computed; in addition, the restriction to one variable does not have violent effects on asymptotic program running times [14,2,24]. We follow this principle in our design of our language *LOOP*, admitting at most a single variable, `X`, to be accessed by any program.

Specialisation and loop unfolding Specialisation has two dimensions: data and control. The data aspect is classically handled using a *division*: a classification of the store (the program's run-time data) into static parts and dynamic parts. In the present context where the store contains only one variable, the division identifies each data operation on a part of the current value of `X` as either *static*: to be computed at specialisation time; or *dynamic*: to be used to generate residual code. The main specialisation technique for data amounts to large-scale constant propagation. We use an analogous technique, adapted to tree-structured data.

The control aspect is more tricky. In most of the specialisers in [18] (the exception being lambda-mix) a program point in the residual program is a pair (pp, vs) where pp is a program point in the subject program, and vs is a tuple of static data values. Any control transfer from a program point pp is specialised into residual form: `goto` (pp, vs), i.e., an unstructured `goto` statement.[3]

Alas, such a solution is incompatible with our goal of efficient interpretation, as it requires considerable program-dependent overhead to interpret a `goto` statement. Hence we restrict the *LOOP* language to structured loops.

A tricky point about specialisation: write `subject,...` $\Rightarrow$ `residual` to mean that a subject command, in the presence of specialisation-time information `...` about static values, is transformed into a residual command. Then the following is a plausible *but incorrect* transformation rule:

$$\frac{\texttt{E},\ldots \Rightarrow \texttt{E}' \quad \texttt{C},\ldots \Rightarrow \texttt{C}'}{\texttt{while E do C},\ldots \Rightarrow \texttt{while E}' \texttt{ do C}'}$$

Interestingly, this familiar-looking context rule is incorrect for specialisation. When execution reaches the end of the loop body `C`, static data may not have

[3] The specialised program points (pp, vs) are kept track of during specialisation time by means of the "pending list", implementing the set `polv` ([18], e.g., Section 4.5).

the same values as they had at the entry to the loop. Thus in the residual (specialised) program the loop does not return to the same point as in the subject program (!)

To see the need for care, consider a simple imperative language with a while-loop and the following example where variable `s` is static and `d` is dynamic.

```
s := 1;
while( d <> 0 ) do
  { if( s = 1 )
    then C1; s := 2
    else if( s = 2 )
         then C2; s = 0
         else C3
  }
```

Since s is statically determined all the `if`s can be handled at specialisation time. Following the inference rule above the code would be specialised to

```
while( d <> 0 ) do C1
```

This is clearly wrong as no account is taken of the changing value of s.

A solution: apply the context rule yielding `while` $\mathtt{E}'$ `do` $\mathtt{C}'$ only if we are *certain* that static values are the same at entry to and exit from $\mathtt{C}'$. Technically this may be done by annotating each "repeat point" (a program point in the loop where control is returned to the beginning of the loop) as residual (not to be unfolded) only when this property holds.

An added benefit of an explicit representation (and annotation) of repeat points is that we may allow unfolding of loops for which static data get smaller (see also [26]), and[18] Section 5.5.1) and that we may avoid both general `goto`s and the "pending list" as a specialiser can be guided by the now-explicit control flow.

The imperative language LOOP and its self-interpreter. Guided by the remarks above, our subject programming language has programs with a single variable X; separated control and data flow; and explicit loop returns. The result is computationally and efficiency-wise equivalent (up to small constant factors) to the usual WHILE language of [16], but easier to manage and analyse.

LOOP syntax is as follows (C = command, E = expression, V = value). Values are binary trees with only a single atom `N` (pronounced "nil"), for example, v = `(N,(N,N))`. Operators: `cons` builds a tree, `hd` and `tl` deconstruct, `=?` tests for equality, and value `N` is read as falsity in `if` and (the result of) `=?`. The single program variable is called `X`.

$C ::= \mathtt{X\ :=}\ E \mid C_1\ \mathtt{;}\ C_2 \mid \mathtt{loop}\{C\} \mid \mathtt{if(}E\mathtt{)\ then}\{C_1\}\ \mathtt{else}\{C_2\} \mid \mathtt{repeat}$

$E ::= \mathtt{X} \mid V \mid \mathtt{hd}\ E \mid \mathtt{tl}\ E \mid \mathtt{cons}\ E_1\ E_2 \mid \mathtt{=?}\ E_1\ E_2$

$V ::= \mathtt{N} \mid (V_1, V_2)$

Semantics: command `repeat` transfers control to the nearest enclosing `loop`; all else is as expected.

The familiar `while` construction can be paraphrased by:

`while E do C` $\equiv$ `loop{if(E)then{C; repeat}else{X:=X}}`

Self-interpretation of LOOP. Self-interpretation is straightforward and details are thus omitted from this short abstract. Constant-overhead interpretation of the `loop{C}` construction is done by pushing `C` on a separate "loop stack", using the stack top when interpreting `repeat`, and popping the stack if control reaches the end of `C` without a `repeat` command. Experiments using an implementation in ML showed that an interpreted program runs around 200 times slower than direct execution when compared using unit cost timing. Symbolically, $time_{\mathtt{sint}}(\mathtt{p},\mathtt{d}) \geq 200 \cdot time_{\mathtt{p}}(\mathtt{d})$.

But the specialiser will be seen to be optimal, so $time_{\mathtt{p}}(\mathtt{d}) \geq time_{\mathtt{sint_p}}(\mathtt{d})$, i.e., *all interpretation overhead is removed* by specialisation. This implies that specialisation gives very high speedup. Indeed, by the definition of Section 1:

$$speedup_{\mathtt{p}}(\mathtt{d}) = \frac{time_{\mathtt{sint}}(\mathtt{p},\mathtt{d})}{time_{\mathtt{sint_p}}(\mathtt{d})} \geq 200$$

3 Specialisation of *LOOP*

An important insight: since the language uses only one variable, the entire store can be represented by a single expression. We can then at specialisation time maintain all static store changes occurring between dynamic store updates efficiently in an "accumulator" expression.

As seen earlier, specialisation of loops is tricky and engenders a need for explicit annotation of "repeat" statements. To cope with this we add to the source program *annotations* that carry information to direct the specialisation.

Two-level annotated LOOP programs We give a very brief account of the 2-level annotation [18, Section 5.3] used, details can be found in [11]. The annotations add a static ($_s$) or dynamic ($_d$) tag to each assignment (`:=`) and conditional (`if`). Tagging a command as dynamic will make the command appear in the residual code. An assignment `:=` is marked static if the the right-hand side of the assignment only involves lookups in parts of `X` known to be static. The dynamic, hence unknown, parts of `X` may be freely copied by giving a reference to their top node in the tree. For example, if the left subtree is dynamic we may statically compute `hd( X )` but not `hd( hd( X ) )`. Conditional commands `if` (E) `then` $\{C_1\}$ `else` $\{C_2\}$ are marked as static if E can be computed at specialisation time.

Expressions may be extended with tag `lift` or `static`. The tag `lift` marks an expression part whose value is static, and will be transferred into the residual program. Tag `static` marks an expression part that does not depend on the

dynamic input.

$$C ::= C_1;C_2 \mid \texttt{loop}\{C\} \mid \texttt{X :=}_s\ E \mid \texttt{X :=}_d\ E \mid \texttt{if}_s(E)\texttt{then}\{C_1\}\texttt{else}\{C_2\} \mid \texttt{if}_d(E)\texttt{then}\{C_1\}\texttt{else}\{C_2\} \mid \texttt{repeat} \mid \texttt{unfold} \mid \texttt{duplicate}$$

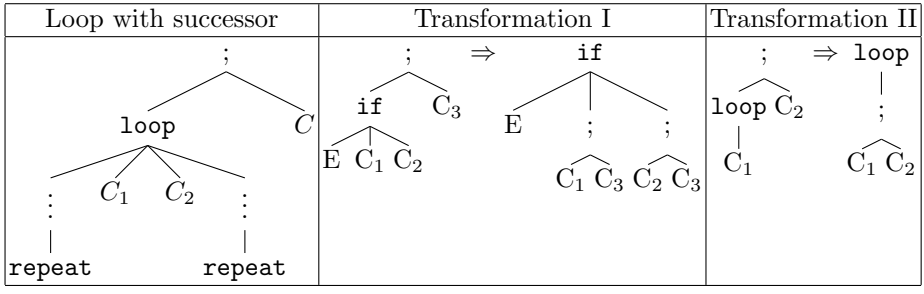
$$E ::= \texttt{X} \mid V \mid \texttt{cons}\ E_1\ E_2 \mid \texttt{=?}\ E_1\ E_2 \mid \texttt{hd}\ E \mid \texttt{tl}\ E \mid \texttt{static}\ E \mid \texttt{lift}\ E$$

$$V ::= \texttt{N} \mid (V_1,V_2)$$

The crux of specialisation of *LOOP* is the `repeat` command; this command may be annotated in one of three ways:

1. Dynamic: generate a residual `repeat`. Annotated form: `repeat`. (As remarked above, this is only semantics preserving if static data are unchanged in the loop; otherwise, incorrect residual code will be produced.)
2. Static duplicate: generate a copy of the entire enclosing loop (not just its body). Annotated form: `duplicate`. This is necessary to handle the interpretation of a `loop` command, which should result in a residual loop. In general this is required when the size of the static data increases.
3. Static unfold: replace `repeat` by the body of the loop. Annotated form: `unfold`. (This is generally semantics-preserving, and terminates if static data has properly decreased in size since the beginning of the loop.)

To enable aggressive specialisation, we perform a number of program transformations to turn the source program into a semantically equivalent program better suited for specialization. Consider the following case and two transformations:

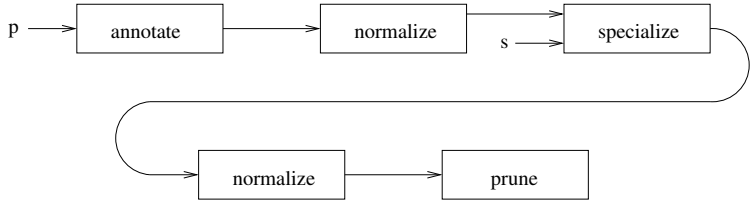


The tricky point is how to specialise the code marked C after specialisation of the loop. A simple program transformation can be performed to turn the source program into an equivalent program where the loop does not have a successor. Concretely, we use a suite of various transformations to alleviate this problem, essentially by moving the code following certain commands like `loop` or `if` *inside* the `loop` or `if` command.

Note that while Transformation I is always applicable, Transformation II may not be performed when execution of C_2 could end in a `repeat`: moving a `repeat` inside a loop may clearly change the semantics of the program, hence should be disallowed.

We denote the process of applying the transformations outlined above (plus numerous terminating small optimisations such as `hd(cons E E′) = E`, etc.) by *normalisation*. The normalisation process has been *formally proven* to terminate; we refer the reader to [11] for details.

The various phases of the specialisation of program p to static data s are:



After the specialiser generates code by a second application of normalisation, further improvements are done. The final phase is pruning. Its purpose: straightforward specialisation may generate "dead" parts of the store that are never referenced by the residual program. Worse, as the store consists of a single variable, these dead parts may cause a *slowdown* in the residual program, since unnecessary store operations may be needed to get to the live parts. To keep the specialisation conceptually simple, we have chosen to perform the removal of dead parts in a separate phase called *pruning*; we again refer the reader to [11] for details. The pruning serves a purpose similar to Romanenko's arity raising [23,22].

Our specialiser performs two tasks while scanning its subject program:

- Generate residual code for dynamic parts of the subject program
- Between points where code is generated, maintain an accumulator expression that sums up the effect of the static program computations since the last point where residual code was generated.

Example To illustrate how the accumulator expression is used to keep track of the static changes to the program, the simple example in the left column of the table below is used. The left branch of X is assumed to be static and the right branch is dynamic. The example uses integers, +, and - as shorthand for easily defined encodings of integers and the successor and predecessor functions.

Annotated subject code	accumulator	Residual code
	X	
X:=s cons(3,X);	cons(3,X)	
X:=s cons(hd(X)+2,tl(X));	cons(5,tl(X))	
X:=d cons(hd(X),tl(X)-1);	X	X:=cons(5,tl(X)-1)
X:=s cons(hd(X)-1,tl(X))	cons(4,tl(X))	
X:=d cons(hd(X),tl(X)-2)	X	X:=cons(4,tl(X)-2)

4 Experimental results

Experiments were performed, divided into a number of *runs*, each representing different configurations of specialisation, with or without involvement of the self-interpreter:

Computer runs I, II, III, IV, V, VI

I: `out := [[p]](s,d)`
II: `out := [[sint]](p,(s,d))`
III: `out := [[sint]]((sint,p),(s,d))`
IV: `out := [[[[spec]](p,s)]](d)`
V: `out := [[[[spec]](sint,p)]](s,d)`
VI: `out := [[[[spec]](sint,sint)]](p,(s,d))`

For each run, a suite of different input programs `p` were considered, in particular, two versions of the append function, a program for lexicographic ordering, and four instances of the string matching problem using a naive string matcher.[4]

The timing results for all experiments are given below. Timing figures count 1 for each primitive operation. Times for runs IV, V, VI are for the outermost $[\![\_]\!]$, i.e., they do not include the time to do specialisation.

Run→ Program↓	I	II	III	IV	V	VI	*Ohead* $\frac{II}{I}$	*Ohead* $\frac{III}{II}$	*Speedup* $\frac{I}{IV}$	*Optim* $\frac{I}{V}$
append	103	19526	4182587	6	103	19526	190	214.2	17.17	1.0
append2	107	20385	4366607	105	107	20385	191	214.2	1.02	1.0
lex	131	24723	5301189	33	131	24723	189	214.4	3.97	1.0
string 1	637	131922	28287715	291	637	131922	207	214.4	2.19	1.0
string 2	21	4121	882810	2	21	4121	196	214.2	10.5	1.0
string 3	115	23682	5077711	55	115	23682	206	214.4	2.09	1.0
string 4	478	98310	21078960	189	478	98310	206	214.4	2.53	1.0

The experimental runs back our claim of optimal specialisation on substantial programs. Specialising programs to static data yields speedups as seen in the *Speedup* column showing the ratio of the execution times of columns I and IV. The column *Optim* shows the relation between the time for direct program execution (I) and the speed of the result of specialising the self-interpreter (V). For the final run (VI) we expect the specialisation of the interpreter with itself to yield an interpreter. Again for optimality this interpreter must run as fast as the original interpreter. The unshown comparison of VI/II shows this to be true.

If the specialiser is optimal the execution time of the specialised interpreter should be at least as fast as direct execution. This is the case for our specialiser. Even though the the specialised interpreter's execution time turned out to be

[4] Ideally the specialisation of the string matcher should produce code equivalent to the Knuth-Morris-Pratt algorithm for string matching. This is not the case here, but the specialisation still provides a substantial speedup.

the same as the time for direct execution, examination of the code produced by the specialiser reveals a program that is not identical to the one the interpreter is specialised to.

Further evidence that we succeeded in our overhead goal is that the interpretive overhead (ratios of columns II/I and III/II) are nearly constant over a wide range of program sizes, even over double self-interpretation (run III).

5 Future work

The semantic basis for specialisation needs to be better formulated, and its correctness proven. Ideally, optimality could be proven (beyond the fairly extensive pragmatic results of the previous section). Further, it would be good to re-express the ideas using a more general store; the constructions we used don't seem to have a fundamental connection with the restriction to one-variable programs.

Another issue concerns program annotation: a *binding-time analysis* has yet to be devised and implemented. The current status is that all test programs were hand annotated (including the self-interpreter, a tricky job). This establishes that the two-level language is expressive enough for nontrivial specialisation. However, a better formal understanding of annotated programs is needed before the process of annotating a program can be completely automated. We aim to do so, once the two-level semantic issues are better understood.

References

1. M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. In *Proceedings of the 2003 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*, pages 3–9. ACM Press, 2003.
2. A. Ben-Amram and N. Jones. Computational complexity via programming languages: Constant factors do matter. *Acta Informatica*, 37:83–120, 2000.
3. D. Bjørner, A. P. Ershov, and N. D. Jones, editors. *Partial Evaluation and Mixed Computation*, North-Holland, 1988. Elsevier Science Publishers B.V.
4. A. Blass and Y. Gurevich. The linear time hierarchy theorems for abstract state machines and rams. *Journal of Universal Computer Science*, 3(4):247–278, apr 1997.
5. C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30(2):79–86, 1989.
6. C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501. ACM Press, 1993.
7. O. Danvy, R. Glück, and P. Thiemann. *Partial Evaluation. Dagstuhl castle, Germany*, volume 1110 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.
8. B. Feigin and A. Mycroft. Jones optimality and hardware virtualization: a report on work in progress. In Hatcliff et al. [12], pages 169–175.
9. J. Gade and R. Glück. On Jones-optimal specializers: A case study using Unmix. In N. Kobayashi, editor, *Programming Languages and Systems. Proceedings*, volume 4279 of *Lecture Notes in Computer Science*, pages 406–422. Springer-Verlag, 2006.

10. R. Glück. Jones optimality, binding-time improvements, and the strength of program specializers. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 9–19. ACM Press, 2002.
11. L. Hartmann. LOOP, a language with Jones optimal interpretation and program independent interpretation overhead. Technical report, DIKU, University of Copenhagen, URL=`ftp://ftp.diku.dk/diku/semantics/papers/D-589.pdf`, 2008.
12. J. Hatcliff, R. Glück, and O. de Moor, editors. *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation, PEPM 2008, San Francisco, California, USA, January 7-8, 2008*. ACM, 2008.
13. J. Hatcliff, T. Mogensen, and P. Thiemann. *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, volume 1706 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
14. N. D. Jones. Constant time factors *do* matter. In S. Homer, editor, *STOC '93. Symposium on Theory of Computing*, pages 602–611. ACM Press, 1993.
15. N. D. Jones. The essence of program transformation by partial evaluation and driving. In M. S. Neil D. Jones, Masami Hagiya, editor, *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu*, pages 206–224. Springer-Verlag, Apr. 1994.
16. N. D. Jones. *Computability and Complexity from a Programming Perspective*. Foundations of Computing. MIT Press, Boston, London, 1 edition, 1997.
17. N. D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52(1-3):307–339, 2004.
18. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., 1993.
19. H. Makholm. On Jones-optimal specialization for strongly typed languages. In W. Taha, editor, *Semantics, Applications and Implementation of Program Generation*, volume 1924 of *Lecture Notes In Computer Science*, pages 129–148, Montreal, Canada, 20 Sept. 2000. Springer-Verlag.
20. T. Mogensen. Inherited limits. In J. Hatcliff, T. Mogensen, and P. Thiemann, editors, *Partial Evaluation: Practice and Theory. Proceedings of the 1998 DIKU International Summerschool*, volume 1706 of *Lecture Notes in Computer Science*, pages 189–202. Springer-Verlag, 1999.
21. J. C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.
22. S. A. Romanenko. Arity raiser and its use in program specialization. In *ESOP 1990*, pages 341–360, 1990.
23. S. A. Romanenko. The specializer UNMIX (for scm scheme). Technical report, Keldysh Institute of Applied Mathematics, Russian Academy of Sciences, 1993.
24. E. Rose. Linear time hierachies for a functional language machine model. In H. R. Nielson, editor, *Programming Languages and Systems – ESOP'96*, volume 1058 of *LNCS*, pages 311–325, Linköping, Sweden, Apr 1996. Linköping University, Springer-Verlag.
25. A. Schönhage. Storage modification machines. *SIAM Journal of Computing*, 9:490–508, 1980.
26. P. Sestoft. Automatic call unfolding in a partial evaluator. In D. Bjørner, A. Ershov, and N. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 485–506. North-Holland, 1988.
27. I. H. Sudborough and A. Zalcberg. On families of languages defined by time-bounded random access machines. *SIAM J. Comput.*, 5(2):217–230, 1976.

28. W. Taha, H. Makholm, and J. Hughes. Tag elimination and Jones-optimality. In O. Danvy and A. Filinski, editors, *Programs as Data Objects*, volume 2053 of *LNCS*, pages 257–275, Heidelberg, Germany, 21–23 May 2001. Springer-Verlag.
29. P. Thiemann. Aspects of the PGG system: Specialization for standard scheme. In *Partial Evaluation - Practice and Theory, DIKU 1998 International Summer School*, pages 412–432. Springer-Verlag, 1999.
30. V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(3):292–325, 1986.