LISP AND SYMBOLIC COMPUTATION: An International Journal, 5, 315–330, 1993

Partial Evaluation in Parallel

CHARLES CONSEL* (*consel@cs.yale.edu*)

Department of Computer Science, Yale University, P.O. Box 2158, New Haven, CT 06520, USA.

OLIVIER DANVY† (*danvy@cis.ksu.edu*)

Department of Computing and Information Sciences, Kansas State University, Manhattan, KS 66506, USA.

Keywords: Specialization of programs, self application, single threading, Scheme, Mul-T, automatic generation of parallel programs, from sequential interpreter to parallel compiler.

Abstract. Partially evaluating a procedural program amounts to building a series of mutually-recursive specialized procedures. When a procedure call in the source program gets specialized into a residual call, the called procedure needs to be processed to occur in the residual program. Because the order of procedure definitions in the residual program is immaterial, it does not matter in which order these two events — building the residual call and building the residual procedure — are scheduled. Therefore, partial evaluation offers a basic opportunity for an MIMD type of parallelism with shared global memory where in essence, the mutually-recursive specialized procedures are built in parallel as specialization points are met and the relation binding source and residual procedures is globalized, to preserve its uniqueness.

We have translated a sequential partial evaluator written in T (a dialect of Scheme) into Mul-T (a parallel extension of T) by adding one semaphore with each specialization point and one future to construct the residual procedure in parallel with the current specialization. The resulting parallel partial evaluator has been observed to be faster than the sequential one in proportion to the size of the source program and to the number of specialized procedures in the residual program.

Our sequential partial evaluator is self-applicable. Because the semaphores and the future are run-time operations, our parallel partial evaluator is still self-applicable. In principle it can be and in practice we have used it to generate parallel compilers, *i.e.*, specializers dedicated to an interpreter and processing its static and dynamic semantics in parallel, non trivially. Again, parallelism in dedicated specializers is determined by the size of the source program and the number of specialized procedures in the residual program.

---

*This work was supported by Darpa under Grant N00014-88-k-0573. Author's new address: Department of Computer Science and Engineering, Oregon Graduate Institute of Science & Technology, 19600 N.W. Von Neumann Drive, Beaverton, OR 97006-1999, USA.

†This work was carried out during a summer visit to Yale University in 1990. This paper was written at Kansas State University and completed during a spring visit at Carnegie Mellon University in 1993.

1. Introduction

This paper describes how a program can be partially evaluated in parallel, based on a simple opportunity for parallelizing a sequential partial evaluator for sequential programs. The attractive properties of partial evaluation are still valid in a parallel setting, in addition to an increased efficiency. In particular, because our partial evaluator is self-applicable, we are able to derive a parallel compiler out of a sequential interpreter, automatically and in parallel. This amounts to improving the "syntax to dynamic semantics" mapping of partial evaluation [8] by making it operate in parallel.

This paper is organized as follows. Section 2 describes what partial evaluation is and how it is implemented. Section 3 points out a very natural opportunity to parallelize partial evaluation and why it is valid. However, parallelization requires globalizing and side-effecting the values that are single-threaded in the sequential partial evaluator. Section 3.1.2 investigates this aspect. Section 3.2 describes how the corresponding synchronization can be refined, thereby minimizing the needs to synchronize. Section 3.3 reports a series of benchmarks. Section 4 addresses how to generate a parallel compiler out of a sequential interpreter. This is achieved by self-application, *i.e.*, by partial evaluation of the partial evaluator with respect to an interpreter. Section 5 compares this approach with related work and puts it into perspective.

2. Partial Evaluation

This section addresses the extensional and intensional aspects of partial evaluation. Partial evaluation specializes programs. It is achieved by executing these programs symbolically.

2.1. Partial evaluation: what

Partial evaluation is a program transformation technique that aims at specializing a program with respect to part of its input, producing a *residual* program. By definition, running a residual program on the remaining input yields the same result as running the source program on the complete input.

$$\begin{aligned} \mathit{run}\ \mathit{PE}\ \langle \mathit{program}, \mathit{incomplete\_input} \rangle &= \mathit{residue} \\ \mathit{run}\ \mathit{residue}\ \langle \mathit{remaining\_input} \rangle &\equiv \mathit{run}\ \mathit{program}\ \langle \mathit{complete\_input} \rangle \end{aligned}$$

Partial evaluation is motivated by the frequent use of a program with constant patterns of input. Specializing this *source* program with respect to such a pattern yields a residual program that expects the remaining variable input to produce the result. A partial evaluator performs those parts

of the source program that depend on the available input. The other computations are performed at run time.

Specialized programs can be expected to be more efficient than source programs since parts of the source program have been executed by the partial evaluator. These static (*i.e.*, partial-evaluation time) computations occur only once, whereas the dynamic (*i.e.*, run time) computations can occur many times.

A broad overview of partial evaluation can be found in the proceedings of recent workshops and symposia [2, 6, 12, 14, 21]. In particular, a complete annotated bibliography of the field is available [2, 12]. This bibliography has been regularly updated ever since by Peter Sestoft in Denmark.[1]

2.2. Partial evaluation: how

Partial evaluation is achieved by executing the program as much as allowed by the available input. The portions of the program that cannot be executed (because they depend on unavailable input) are reconstructed. This process yields a residual, specialized program.

We consider the partial evaluation of Scheme programs [4]. Our source and residual programs are written in Scheme. We consider Consel's partial evaluator Schism [7]. Schism is written in Scheme as well and is self-applicable, *i.e.*, it can specialize itself non-trivially.

A Scheme program is a collection of mutually-recursive procedures. According to some criteria which relate to making partial evaluation terminate [3, 5] but which fall outside the scope of this paper,[2] some procedures are *specialization points* whereas all the others are *unfolded*. A specialization point potentially gives rise to many specialized versions of the source callee in the residual program.

Processing a call to a specialization point amounts to creating a residual call to a specialized procedure in the residual program and to constructing this specialized procedure. These two events are scheduled either breadth-first — the residual call is built right away and the procedure to be specialized is recorded for future treatment; or depth-first — the called procedure is processed right away, the specialized procedure is recorded for future reference, and then the residual call is built.

Due to this latitude in the traversal order, partial evaluation offers a basic opportunity for parallelism.

[1] The most recent version (/pub/doc/partial-eval.bib.Z) is available from ftp.diku.dk by anonymous ftp.

[2] In practice, specialization points are dynamic conditional expressions. The partial-evaluation strategy we consider is offline and performs polyvariant specialization [9, 15].

3. Parallelism

It does not matter in which order residual calls are built and procedures are specialized because the order of procedure definitions in the residual program is immaterial. However parallelizing these constructions requires a synchronization, as addressed in Section 3.1.1. This synchronization requires globalizing and side-effecting what were single-threaded values in the sequential partial evaluator, as addressed in Section 3.1.2. Finally, Section 3.3 reports some measures of the parallel partial evaluator.

3.1. Parallelism: where

Section 2 ended on a basic opportunity for parallelism. Instead of complying with the breadth-first or depth-first strategy, the two actions — building the residual call and specializing the source procedure — can be executed in parallel. Overall, partial evaluation is finished when the collection of residual procedures is completely built.

3.1.1. Synchronization

One synchronization is necessary, though, because two calls to the same specialization point with respect to the same static values should yield a residual call to the same residual procedure — otherwise the partial evaluator would build several identical residual procedures. At best it would give large and redundant residual programs; at worst (which occurs quite often, typically with a recursive residual procedure), it would not terminate.

Let us illustrate this last point with a simple example and specialize the following source program

```
(define main
  ;;; List(A) * List(A) * List(A) -> List(A) * List(A)
  (lambda (x y z)
    (cons (append x z) (append y z))))

(define append
  ;;; List(A) * List(A) -> List(A)
  (lambda (x y)
    (if (null? x)
        y
        (cons (car x) (append (cdr x) y)))))
```

with respect to `z` = `(3 4 5)` and the two other parameters unknown. Because the second parameter of `append` is not its induction variable, we classify `append` as a specialization point. Because this procedure is called three

times with the same static argument during partial evaluation (two times in `main` and one time in `append`), the same residual procedure will be called three times in the residual program:

```
(define main-0
  ;;; List(Num) * List(Num) -> List(Num) * List(Num)
  (lambda (x y)
    (cons (append-1 x) (append-1 y))))

(define append-1
  ;;; List(Num) -> List(Num)
  (lambda (x)
    (if (null? x)
        '(3 4 5)
        (cons (car x) (append-1 (cdr x))))))
```

This specialized program expects the two remaining arguments and computes the same result as the source program, but somewhat faster, since `append-1` is unary whereas `append` was binary. Without synchronization, partial evaluation would loop in the attempt to create infinitely many instances of `append-1`.

This example stresses the need for synchronizing accesses to the mapping between source program points, static values, and the corresponding residual procedures. For each source specialization point, a series of static values should determine a residual procedure uniquely. The following section describes how this uniqueness is preserved in the presence of parallelization.

3.1.2. Single-threaded values and the parallelization of functional programs

Schism [7] is a partial evaluator written in pure Scheme, *i.e.*, it is side-effect free. It keeps a *log* (a.k.a. a *cache*) recording which source procedures have been specialized with respect to which static values. This log is consulted and updated as the source program gets partially evaluated.

The only way to implement the log (consultation and updating) in a purely functional program is to pass it along to all the potential users and receive it updated from them, much like the store in the denotational specification of an imperative language. Since only one copy exists at any time during execution, the log is single-threaded [20]. In practice, single-threaded values are bound to global variables that are side-effected, to achieve efficiency while benefiting from the functional framework.

Single-threaded values assume a unique execution thread. Therefore, in general, they make it impossible to parallelize a program, since by definition, parallelizing amounts to creating several execution threads.

However, our log does not need to be updated sequentially. It only requires having a unique representation at any time, as this representation gets updated. It does not matter where and when a specialization point was first encountered with a particular set of static values. Simply put, all the later encounters to this specialization point with the same set of static values should produce a residual call to the same specialized procedure. The first encounter triggers the specialization of a source procedure. All the other encounters refer to the specialized version of this procedure. Therefore, to parallelize Schism, we can globalize the log and synchronize its consultation and update using a semaphore. However, in contrast to the sequential case, side-effecting the log is not an optimization any more — it is a requirement.

This experiment illustrates how a functional program with single-threaded values can be parallelized when the order of access and update to these values does not matter. After globalization, the operations over the single-threaded value become semaphore operations that are essential to the correctness of the parallelized program.

In conclusion, sequential partial evaluation offers an opportunity for an MIMD type of parallelism with shared global memory.

3.2. Parallelism: how

Schism is written in the T dialect of Scheme [19]. T has been extended with Dijkstra's semaphores [11] and Halstead's futures [1] in the Mul-T system [17]. In spirit, this extension encourages one to spread a few parallel constructs in an existing sequential program to get a parallel one. Task scheduling is automatic [18]. Let us describe how to parallelize Schism.

Parallelizing the breadth-first or depth-first traversal of the source program is achieved using a future construct. In the original partial evaluator, the construction of the residual call and the specialization of the callee are sequentialized. In the new partial evaluator, we parallelize the construction of the residual call and the specialization of the callee. We do this by building a promise to specialize the callee (this will yield the residual procedure) and by constructing the residual call right away.

Synchronizing accesses and updates to the log is achieved with a semaphore. This makes it possible to preserve the uniqueness of the relationship between source procedures, static values, and residual procedures that was ensured, in the sequential case, with a single-threaded log.

However, this synchronization is too coarse, and as such it strangles the whole partial-evaluation process, since there is no need to synchronize when two distinct specialization points are reached. Synchronization only matters

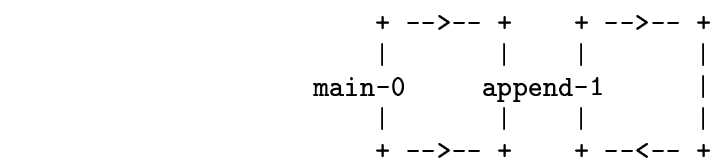

Figure 1: Task graph

when the same specialization point is reached concurrently. Therefore, we refine the synchronization by associating one semaphore with each source specialization point. (We did not have the opportunity to compare the uses of a single semaphore *vs.* the use of multiple semaphore, though.)

To summarize, we parallelize Schism by wrapping a future around the construction of a residual procedure and by associating one semaphore with each specialization point.

One can draw the dependency graph of the parallel processes during partial evaluation, making a node stand for each task constructing a residual procedure and one vertex stand for each access to the log. Figure 1 displays the task graph corresponding to the example above. Processing `main-0` spawns the processing of `append-1` that would spawn the processing of `append-1` if this was not done already.

As can be noticed, this graph is identical to the call graph of the residual program (before trivial post-unfolding [3]). It has been built in parallel as the static components of the source program were processed in parallel. Accordingly, we can expect source programs to specialize faster if they have many specialization points and if many computations only depend on the available input, because these computations can be performed in parallel.

3.3. Benchmarks for parallel partial evaluation

This section presents the results of specializing a program in parallel.

$$run\ PE_p\ \langle program, incomplete\_input \rangle \quad = \quad residue$$

PE_p is the parallel partial evaluator. We have applied it to a pattern matcher for lists [10], an interpreter for the applied λ-calculus, and the specializer.

The first column of Table 1 shows the number of specialization points for each program; this component determines the degree of parallelism that occurs when the corresponding program is specialized (recall that a task is created for each procedure specialization). The second column of Table

Applications	Nb of Spec. Points	Size (Chars)	Max. Speedup
pattern matcher	1	3906	1.04
λ-interpreter	2	4865	1.14
specializer	21	57626	3.29

Table 1: Speedups of parallel partial evaluation with 8 processors over sequential partial evaluation

Number of Processors	1	2	4	8
Run time	6.94	6.32	6.19	6.10
Number of Futures	2	2	2	2
Idle cycles	0	32671	99108	229593
Speedup	1.00	1.10	1.12	1.14

Table 2: Specialization of the λ-interpreter

1 displays the size of each program; it provides a rough estimate of the granularity of each task — assuming procedures have comparable size and an equal amount of static computations, on average. The third column displays the maximum speedup obtained by the parallel partial evaluator, using the eight processors available, over the sequential partial evaluator on one processor; it demonstrates the importance of the specialization points in speeding up the specialization process — few specialization points yield a poor parallelism and thus a negligible speedup. Fortunately, for more realistic programs like the specializer, more specialization points produce a better parallelism which accelerates the specialization process considerably (*e.g.*, more than three times for the specializer).

In the rest of this section, we examine the two first applications further. The third experiment is delayed until Section 4 that addresses self-application. Unfortunately, we ran out of time to carry out more experiments.

Tables 2 and 3 display the performance statistics for the specialization of the λ-interpreter and the pattern matcher with respect to a variety of λ-programs and of patterns, respectively. In both cases, specialization does not improve when the number of processors is increased. This is due to the poor latent parallelism of these applications: specializing the λ-interpreter only yields two residual procedures, and only one for the pattern matcher. Two futures have been created for the former; only one has been created

Number of Processors	1	2	4	8
Run time	5.22	5.01	5.09	5.01
Number of Futures	1	1	1	1
Idle cycles	0	27177	82762	189866
Speedup	1.00	1.04	1.02	1.04

Table 3: Specialization of the pattern matcher

for the latter because there is only one specialization point and its access is locked by the semaphore (*cf.* second line of Tables 2 and 3). The number of futures does not include the main task that initiates the specialization.

The run-time figures are given in seconds with two decimals. They were obtained on the Encore Multimax machine with the version 4.0 of the Mul-T compiler [17]. They exclude the time used for garbage collection and have the usual uncertainty connected to CPU measures.

3.4. Conclusion

Overall, the tables above confirm our initial intuition that parallel partial evaluation performs poorly for small programs with few specialization points. Let us now consider a real size example — the specializer — and examine in detail better results yielded by the parallel strategy.

4. Self-Application

Self-applying a partial evaluator aims at optimizing the process of specialization. In a nutshell, repeated partial evaluation of a program with respect to different values can be optimized first by generating a specializer dedicated to this program and second by running this dedicated specializer repeatedly. This is captured in the following equations.

A partial evaluator specializes programs:

$$run\ PE\ \langle program, data\rangle \quad = \quad residue$$

Partial evaluation can be optimized by preliminary generation of a dedicated specializer:

$$\begin{aligned} run\ PE\ \langle PE,\ \langle program, \_\rangle\rangle &= DS \\ run\ DS\ \langle data\rangle &= residue \end{aligned}$$

Partial evaluation can be optimized further by preliminary generation of a generator of dedicated specializers:

$$\begin{aligned} run\ PE\ \langle PE,\ \langle PE, \_\rangle\rangle &= Curry \\ run\ Curry\ \langle program, \_\rangle &= DS \end{aligned}$$

Curry takes its name from the fact that it curries a program intensionally. In other words, it may take a two-argument program and return a new program expecting the first argument and yielding another program that expects the second argument and produces the result of the original program if the original program is applied to both arguments. This transformation is intensional because it operates on the textual representation of these programs.

Self-application has received a great deal of attention due to the following particular case. If the source program is an interpreter, the dedicated specializer has the functionality of a compiler [13]. Correspondingly, if the source program is the partial evaluator itself, applying the dedicated specializer to an interpreter makes it have the functionality of a compiler generator [16].

Our sequential partial evaluator is self-applicable. Now that we have a parallel partial evaluator, how can we self-apply it and what does self-application yield? The following sections answer these two questions.

4.1. Self-application in parallel

Self-application is only the particular case where the source program is the partial evaluator itself. Therefore, the sequential partial evaluator can be specialized in parallel, yielding the same sequential dedicated specializer, modulo the order of procedure definitions:

$$run\ \ PE_p\ \ \langle PE,\ \langle program,\ \_\ \rangle\rangle\ \equiv\ \ run\ \ PE\ \ \langle PE,\ \langle program,\ \_\ \rangle\rangle$$

The parallel apparatus described in Section 3.2 aims at making promises about the definition of residual procedures and at synchronizing the access to the global log. Both these operations depend on the static values a program is specialized with respect to. Because these values are not available at self-application time, these operations need to be classified as dynamic, *i.e.*, they will be performed (in some undetermined order) only when the dedicated specializer runs.

Therefore, by declaring the semaphore operations and the future construction to be dynamic algebraic operators, we can specialize the parallel partial evaluator, either sequentially or in parallel:

$$run\ \ PE\ \langle PE_p,\ \langle program,\ \_\ \rangle\rangle\ \equiv\ \ run\ \ PE_p\ \langle PE_p,\ \langle program,\ \_\ \rangle\rangle$$

This illustrates how, in a sense, a partial evaluator only needs to know about the static subset of its input language.

4.2. Parallel dedicated specializers

Because the operations managing parallelism are deferred until the dedicated specializer runs, self-application produces residual programs with parallelization points. In other words, dedicated specializers run in parallel as well.

To summarize, parallel partial evaluation can be optimized by preliminary parallel generation of a dedicated, parallel specializer:

$$\begin{aligned} run\ PE_p\ \langle PE_p,\ \langle program, \_\rangle\rangle &= DS_p \\ run\ DS_p\ \langle data\rangle &= residue \end{aligned}$$

Parallel partial evaluation can be further optimized by preliminary generation of a parallel generator of dedicated, parallel specializers:

$$\begin{aligned} run\ PE_p\ \langle PE_p,\ \langle PE_p, \_\rangle\rangle &= Curry_p \\ run\ Curry_p\ \langle program, \_\rangle &= DS_p \end{aligned}$$

In the particular case of interpreters, we obtain parallel compilers that are generated in parallel. Let us compare them and their generation with their sequential counterparts.

4.3. Benchmarks for parallel self-application

The following measures have been taken under the same conditions as in Section 3.3. They account for the results of generating a specializer dedicated to a program.

$$run\ PE_p\ \langle PE_p,\ \langle program, \_\rangle\rangle\ =\ DS_p$$

Tables 2 and 3 display the performance statistics for self-application with respect to the λ-interpreter and the pattern matcher, respectively — *i.e.*, timings for compiler generation. As can be observed, these applications improve when more processors are provided. With eight processors these applications are two to three times faster than with one processor.

Number of Processors	1	2	4	8
Run time	267.00	142.51	93.25	81.08
Number of Futures	21	21	21	21
Idle cycles	0	105092	570621	2049373
Speedup	1.00	1.87	2.86	3.29

Table 4: Self-application with respect to the λ-interpreter

Number of Processors	1	2	4	8
Run time	416.02	223.64	165.84	161.45
Number of Futures	24	24	24	24
Idle cycles	0	142327	1292846	4664247
Speedup	1.00	1.86	2.51	2.58

Table 5: Self-application with respect to the pattern matcher

4.4. Conclusion

Overall, the dedicated specializers run faster than the corresponding direct partial evaluation. The likelihood of this improvement originally motivated self-application [13, 16]. We have observed that in all cases, the improvement of DS_p over DS is similar to the improvement of PE_p over PE (*cf.* Table 1). However, from the parallelism viewpoint, self-application does not change anything: as confirmed by experience, the parallelism shown in specializing the pattern matcher or the λ-interpreter is the same as the one of the dedicated specializers.

5. Conclusion and Issues

We have identified a simple opportunity for parallelism in a partial evaluator for procedural programs. We have implemented it by extending an existing sequential partial evaluator, and we have measured its effect on typical specializations. We have pointed out how to make this parallelization compatible with self-application and we have generated parallel specializers dedicated to sequential programs. Actually, and based on the current partial evaluation bibliography [9, 15], we are not aware of any other self-applicable partial evaluator operating in parallel.

From the point of view of partial evaluation, this experiment was worth-

while. Our parallel partial evaluator is a conservative extension of the sequential one and runs faster in general. The same applies for compilers and other dedicated specializers generated by self-application.

From the point of view of parallelism, let us keep a lower profile. The present opportunity for parallelism — subsuming breadth-first and depth-first traversals — is a basic one. It relies on a particular model of parallelism — MIMD with shared global memory with a coarse-grained, small-scale parallelism (the number of synchronization points is fixed for each source program; the number of futures is determined by the static input). On the other hand, specializing a program in parallel appears to be new, and of course so does the generation of parallel dedicated specializers in parallel, since it requires mastering self-applicable partial evaluation.

This experiment confirms other experiments based on Mul-T — programs with a good potential for parallelism perform better. It also confirms our original thesis — exploiting this latent parallelism of a partial evaluator should optimize partial evaluation when many procedures get specialized. Applications to other languages than Scheme require a corresponding partial evaluator, which is the goal of current research [2, 6, 12, 14, 21].

The degree of parallelism obtained partly depends on the source program that gets specialized, *i.e.*, on how many specialization points are reached in parallel. This suggests a possible control of this parallelism, for example using a static analysis, to determine the order in which a program should be specialized, *i.e.*, the order in which specialization points should be reached. This would make it possible to constrain and thus to regulate the degree of parallelism during partial evaluation.

Regarding the specialization of programs with explicit parallelism, we took the drastic option to delay the parallel operations until run time in Section 4 because they depend on dynamic data anyway. Partial evaluation of general parallel programs would probably require one to distinguish between static parallelism (the source program computations performed by the partial evaluator) and dynamic parallelism (the remaining computations performed in the residual program). In the particular case of self-application, there is no static parallelism in this sense.

As another issue, one may consider other opportunities for parallelizing partial evaluation, and whether they are as natural as this Gordian treatment of depth-first versus breadth-first traversals. We leave this to a future work.

Acknowledgements

Rick Mohr pointed out how crucial it is to globalize and side-effect a single-threaded value when parallelizing a functional program. We are grateful to the referees and also to Andrzej Filinski, Karoline Malmkjær, Austin Melton, Dave Schmidt, Carolyn Talcott, and Virg Wallentine for their perceptive and thoughtful comments.

References

1. Halstead, Jr., Robert H. MultiLisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7, 4 (1985) 501–538.

2. Bjørner, Dines, Ershov, Andrei P., and Jones, Neil D., editors. *Partial Evaluation and Mixed Computation*. North-Holland (1988).

3. Bondorf, Anders and Danvy, Olivier. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16 (1991) 151–195.

4. Clinger, William and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV, 3 (July-September 1991) 1–55.

5. Consel, Charles. *Analyse de Programmes, Evaluation Partielle et Génération de Compilateurs*. PhD thesis, Université Pierre et Marie Curie (Paris VI), Paris, France (June 1989).

6. Consel, Charles, editor. *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, San Francisco, California (June 1992).

7. Consel, Charles. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, Copenhagen, Denmark (June 1993). To appear.

8. Consel, Charles and Danvy, Olivier. Static and dynamic semantics processing. In *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, Orlando, Florida (January 1991) 14–24.

9. Consel, Charles and Danvy, Olivier. Tutorial notes on partial evaluation. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM Press, Charleston, South Carolina (January 1993) 493–501.

10. Danvy, Olivier. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37 (March 1991) 315–322.

11. Dijkstra, Edsger W. Co-operating sequential processes. In Genuys, F., editor, *Programming Languages*, Academic Press (1968) 43–112.

12. Ershov, Andrei P., Bjørner, Dines, Futamura, Yoshihito, Furukawa, K., Haraldsson, Anders, and Scherlis, William, editors. *Special Issue: Selected Papers from the Workshop on Partial Evaluation and Mixed Computation, 1987 (New Generation Computing, vol. 6, nos. 2,3)*, Ohmsha Ltd. and Springer-Verlag (1988).

13. Futamura, Yoshihito. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls 2, 5* (1971) 45–50.

14. Hudak, Paul and Jones, Neil D., editors. *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, New Haven, Connecticut (June 1991).

15. Jones, Neil D., Gomard, Carsten K., and Sestoft, Peter. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall International (1993). To appear.

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

17. Mohr, Eric. *Mul-T*. Yale University, New Haven, Connecticut, USA (1990). Release Notes.

18. Mohr, Eric, Kranz, David A., and Halstead Jr., Robert H. Lazy task creation: A technique for increasing the granularity of parallel programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, ACM Press, Nice, France (June 1990) 185–197.

19. Rees, Jonathan A., Adams, Norman I., and Meehan, James I. *The T Manual*. Yale University, New Haven, Connecticut, USA (1984). Fourth edition.

20. Schmidt, David A. Detecting global variables in denotational definitions. *ACM Transactions on Programming Languages and Systems*, 7, 2 (April 1985) 299–310.

21. Schmidt, David A., editor. *Proceedings of the Second ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, ACM Press, Copenhagen, Denmark (June 1993). To appear.