BRICS RS-99-45 T. Amtoft: Partial Evaluation for Constraint-Based Program Analyses

BRICS

Basic Research in Computer Science

Partial Evaluation for Constraint-Based Program Analyses

Torben Amtoft

BRICS Report Series **RS-99-45**

ISSN 0909-0878 **December 1999**

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

Partial Evaluation for Constraint-Based Program Analyses

Torben Amtoft *

December, 1999

Abstract

We report on a case study in the application of partial evaluation, initiated by the desire to speed up a constraint-based algorithm for control-flow analysis. We designed and implemented a dedicated partial evaluator, able to specialize the analysis wrt. a given constraint graph and thus remove the interpretive overhead, and measured it with Feeley's Scheme benchmarks. Even though the gain turned out to be rather limited, our investigation yielded valuable feed back in that it provided a better understanding of the analysis, leading us to (re)invent an incremental version. We believe this phenomenon to be a quite frequent spinoff from using partial evaluation, since the removal of interpretive overhead makes the flow of control more explicit and hence pinpoints sources of inefficiency. Finally, we observed that partial evaluation in our case yields such regular, low-level specialized programs that it begs for runtime code generation.

1 Introduction

Offline partial evaluation benefits much from static program analysis [5, 9], but could the converse be true as well? In this work, we consider a "typical" program analysis, essentially similar to the constraint-based control flow analysis described in Nielson, Nielson, and Hankin's textbook on program analysis [12, Chap. 3], and use partial evaluation (a.k.a. program specialization) to eliminate the interpretive overhead incurred by the analysis.

Control flow analysis (CFA) is an integral part of any analysis of higher-order programs. In the 1990's, the trend—pioneered by Palsberg & Schwartzbach [13] and continued by, e.g., Henglein [8]—has been to take a constraint-based approach to flow analysis, so as to avoid iterating over (complex) source terms and instead traverse a graph built from a set of constraints. (Gasser, Nielson & Nielson's ICFP'97 paper [7] illustrates how to convert an abstract specification

*This work was initiated at BRICS in the fall of 1998, and continued in the Church project with the support of NSF Grant EIA-9806747. Current address of the author: Department of Computer Science, Boston University, 111 Cummington Street, Boston, MA 02215, USA. E-mail: tamtoft@cs.bu.edu.

of a flow analysis into a form implementable by constraints.) The virtue of this approach, it is often said, is that "syntax disappears." In the sense that the analysis then does not depend on the peculiarities of the syntax of a particular language, this claim is true. But in the sense that all interpretation overhead is removed, this claim is actually false: the constraint graph is repeatedly examined and traversed until all constraints are satisfied. In Sect. 2 we shall see that it is possible to produce constraint solvers that are specialized wrt. a given control graph, and which therefore run without interpretive overhead.

Program specialization amounts to constant propagation together with unfolding, and is therefore able to reduce run time by at most a constant factor [1, 9]. Moreover, unless we are dealing with separate compilation (in which case the initial condition of the "interface nodes" may vary), the residual program will be run once only. One may therefore ask whether it really pays off to specialize a control flow analysis, since the cost of generating the residual program may easily outweigh the gain in run time.

The answer is that eventually (for very large programs) it probably will, as generation time grows asymptotically slower than run time (cf. the complexity estimates in Sects. 2.2 and 3.1). Unfortunately, our current results (Sect. 4) do not allow us to report on such a success story.

In another respect, however, partial evaluation turned out to beneficial also in our case: it provided feedback on the control flow analysis presented in Sect. 2, in that the structure of the residual code highlighted the inherent inefficiency of the algorithm. As reported in Sect. 3, this led us to the design of an incremental version with much better asymptotic behaviour (cf. Sect. 3.1).

The resulting algorithm is by no means new. Nevertheless, we believe our case study indicates that a main virtue of doing partial evaluation is that it assists the invention of clever and efficient algorithms. Support for this claim has also been provided by Consel and Danvy [4] in the context of string matching.

2 CFA by solving constraints

We now embark on presenting our (initial) algorithm for control flow analysis. As input the analysis takes a Scheme program, where each subexpression and each bound variable is assigned a unique integer label—thus we can identify subexpressions with their labels. (We shall use the letters l, p and q to range over labels.) The program *idid*, depicted below, will serve as our running example.

$$\texttt{((lambda (x}^{2}\texttt{) x}^{4}\texttt{)}^{3}\ \texttt{(lambda (x}^{5}\texttt{) x}^{7}\texttt{)}^{6}\texttt{)}^{1}$$

The output of the analysis is a mapping D assigning a set of labels to each label, with the following (informally stated) property: for each subexpression e (respectively bound variable x) in the source program, if e at runtime may evaluate to (respectively x may denote) a closure (l, E) (with l a lambda abstraction and E some environment) then $l \in \mathsf{D}(e)$ (respectively $l \in \mathsf{D}(x)$). This property can be formalised by giving a small-step operational semantics for our Scheme subset, and then establishing a “flow preservation” property. This would be much similar to what is done in, e.g., Nielson and Nielson’s POPL’97 paper [11].

We now consider our running example to see what will constitute a valid D. Lambda abstractions evaluate to themselves, so we can take $\mathsf{D}(3) = \{3\}$ and $\mathsf{D}(6) = \{6\}$. By β-reduction $\mathtt{x}^2$ will be bound to $\mathtt{(lambda\ (x^5)\ x^7)}^6$, showing that $\mathsf{D}(2) = \{6\}$ is the least possible choice for $\mathsf{D}(2)$. Hence also $\mathsf{D}(4) = \{6\}$, implying that $\mathsf{D}(1) = \{6\}$ will do the job (as *idid* itself evaluates to $\mathtt{(lambda\ (x^5)\ x^7)}^6$). On the other hand, $\mathtt{(lambda\ (x^5)\ x^7)}^6$ is never applied so we can safely take $\mathsf{D}(5) = \mathsf{D}(7) = \emptyset$.

It is pretty straightforward to convert a source program into a set of *constraints* on the value of D, such that any solution will indeed satisfy the correctness property stated above. The only interesting case is the one for applications $\mathtt{(l_0\ l_1\ \ldots\ l_n)}^l$ where *conditional* constraints are needed: for each lambda expression $\mathtt{(lambda\ (l'_1\ \ldots\ l'_n)\ l')}^{l'_0}$ occurring in the whole program (note that we only consider those whose arity match), we generate the constraints

$$\begin{array}{rl} \left.\begin{array}{r} l'_0 \in \mathsf{D}(l_0) \Rightarrow \mathsf{D}(l_1) \subseteq \mathsf{D}(l'_1), \ldots \\ l'_0 \in \mathsf{D}(l_0) \Rightarrow \mathsf{D}(l_n) \subseteq \mathsf{D}(l'_n) \end{array}\right\} & \text{actuals flow to formals} \\ l'_0 \in \mathsf{D}(l_0) \Rightarrow \mathsf{D}(l') \subseteq \mathsf{D}(l) \qquad\qquad & \text{body flows to result} \end{array}$$

Our example program *idid* gives rise to the constraints depicted below, and it is easy to check that the D tabulated above is in fact the least solution of these constraints.

$$\begin{array}{ll} \{3\} \in \mathsf{D}(3),\ \{6\} \in \mathsf{D}(6), & \mathsf{D}(2) \subseteq \mathsf{D}(4),\ \mathsf{D}(5) \subseteq \mathsf{D}(7), \\ 3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(2), & 3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(4) \subseteq \mathsf{D}(1) \\ 6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(5), & 6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(7) \subseteq \mathsf{D}(1) \end{array}$$

We now address the task of our constraint-based flow analysis: to compute the least solution of a set of constraints. For that purpose we use the algorithm presented in Nielson, Nielson and Hankin’s textbook on program analysis [12, Chap. 3], our SML-implementation of which is listed[1] in Fig. 1.

The solution D is computed by iteration, controlled by a work list W which initially contains those q for which there is a constraint $\{q_0\} \in \mathsf{D}(q)$. Once an element q is extracted from the work list it is examined by the function `iter`, and for all q' such that $\mathsf{D}(q)$ has to be subset of $\mathsf{D}(q')$ the function `addD` is called on q' and $\mathsf{D}(q)$ so as to if necessary augment $\mathsf{D}(q')$—if so, q' is added to W so that the updates can be further propagated.

To ensure that one from q can easily find the relevant q' to update, the function `process_ccs` has constructed a graph E with the program labels as nodes. An unconditional constraint $\mathsf{D}(q) \subseteq \mathsf{D}(q')$ has to be reexamined each time q is updated so we add this constraint to $\mathsf{E}(q)$, in effect creating an edge from q to q'. A conditional constraint $l \in \mathsf{D}(p) \Rightarrow \mathsf{D}(q_1) \subseteq \mathsf{D}(q_2)$ has to be

[1] Here `NodeIncl (q1,q2)` denotes the constraint $\mathsf{D}(q1) \subseteq \mathsf{D}(q2)$, `DataIncl (l,p)` denotes the constraint $\{l\} \in \mathsf{D}(p)$, and `CondNodeIncl (l,p,q1,q2)` denotes the constraint $l \in \mathsf{D}(p) \Rightarrow \mathsf{D}(q1) \subseteq \mathsf{D}(q2)$. Sets of labels are manipulated by the operations prefixed by `Data`, such as `empty`, `union`, `is_subset`, `print`. The program uses arrays which are initialized by `ArrI`, updated by `ArrU`, and examined by `ArrL`.

```
fun CFA1 (size,ccs) = let
 val E = ArrI (size,[]:Constr list)
 fun addE q cc = ArrU (E,q,cc::(ArrL (E,q)))
 val W = ref ([]:int list)
 val D = ArrI (size,Data.empty)
 fun addD q d = let val Dq = ArrL (D,q)
                in if Data.is_subset (d,Dq) then ()
                   else ( ArrU (D,q, Data.union (d,Dq))
                        ; W := (q::(!W))) end
 fun process_ccs [] = ()
 |   process_ccs ((cc as NodeIncl (q1,q2))::ccs) =
             (addE q1 cc; process_ccs ccs)
 |   process_ccs (DataIncl (l,p) :: ccs) =
             (addD p (Data.mk_singleton l); process_ccs ccs)
 |   process_ccs ((cc as CondNodeIncl (l,p,q1,q2)) :: ccs) =
             (addE q1 cc; addE p cc; process_ccs ccs)
 val _ = process_ccs ccs

 fun iter q = iter' (ArrL (E,q))
 and iter' [] = iterate ()
 |   iter' (NodeIncl (p1,p2)::ccs) =
             (addD p2 (ArrL (D,p1)); iter' ccs)
 |   iter' (CondNodeIncl (l,p,p1,p2)::ccs) =
             ( (if Data.is_member (l,ArrL (D,p))
                then addD p2 (ArrL (D,p1)) else ())
             ; iter' ccs)
 and iterate () = case !W of [] => ()
                 | (q::W') => (W := W'; iter q)
 val _ = iterate ()
in fn q => (print ((Data.print (ArrL (D,q)))^"\n"); ())  end
```

Figure 1: The constraint-solving approach: the interpreter

reexamined each time p or q_1 is updated so we add this constraint to $\mathsf{E}(p)$ and $\mathsf{E}(q_1)$, in effect creating edges from p to q_2 and from q_1 to q_2.

For our example program *idid*, W initially contains 3 and 6 with $\mathsf{D}(3) = \{3\}$ and $\mathsf{D}(6) = \{6\}$. The value of E is depicted below:

$$\mathsf{E}(2) = \{\mathsf{D}(2) \subseteq \mathsf{D}(4)\} \qquad \mathsf{E}(5) = \{\mathsf{D}(5) \subseteq \mathsf{D}(7)\}$$
$$\mathsf{E}(4) = \{3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(4) \subseteq \mathsf{D}(1)\} \qquad \mathsf{E}(7) = \{6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(7) \subseteq \mathsf{D}(1)\}$$
$$\mathsf{E}(6) = \{3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(2),\ 6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(5)\}$$
$$\begin{aligned}\mathsf{E}(3) = \{&3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(2),\ 3 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(4) \subseteq \mathsf{D}(1),\\ &6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(6) \subseteq \mathsf{D}(5),\ 6 \in \mathsf{D}(3) \Rightarrow \mathsf{D}(7) \subseteq \mathsf{D}(1)\}\end{aligned}$$

2.1 Removing interpretation overhead

The constraint-based flow analysis in Fig. 1 contains some interpretation overhead. In particular, `iter` has to examine the constraints in E in order to find out which action to take. It would be preferable if this "next action" were encoded as control, rather than as data.

It is in fact possible to achieve this goal: we have written a dedicated partial evaluator (a "generating extension" [9]) that from a constraint set produces a residual program where this layer of interpretation has been removed. To be more precise, we classify as static the construction of E by `process_ccs` and the dispatch on the value of $\mathsf{E}(q)$ by `iter`. On the other hand, e.g., all calls to `addD` are classified as dynamic and are therefore present in the residual program.

As the generating extension basically operates by evaluating the static operations and producing code for the dynamic operations, one can easily be convinced of its correctness: the residual program will evaluate to the same result as does the interpreter when applied to the constraints.

The result of applying the generating extension to *idid* is depicted[2] in Fig. 2. As desired, this residual program evaluates without interpretation overhead: the edges in E have been "hard-wired" into the code. Nevertheless, one can spot some other sources of inefficiency; we will return to that issue in Sect. 3.

2.2 Complexity estimates

We now informally provide some complexity bounds, for the interpreter as well as for the generated residual programs. We shall assume that all operations associated with `Data` can be done in constant time, and do not count the time used for converting the source program into a set of constraints.

Given a source program, we let N denote its size (i.e., the number of labels), we let A denote the number of function applications, and we let L denote the number of lambda abstractions. With these entities as parameters, we want to estimate

- I, the time spent by the interpreter when applied to the constraints generated from the source program;
- S, the size of the residual program produced by the generating extension;
- and R, the time required to run this residual program.

First observe that the number of constraints is in $O(N + AL)$ (assuming that there is a maximal arity for functions), and as each constraint appears at most twice in E also the size of E is in $O(N + AL)$. From this we infer that also S is in $O(N + AL)$. Using only N as parameter, we thus have $S \in O(N^2)$.

To give a tight estimate of R, we introduce an extra entity H denoting the maximal size of any $\mathsf{D}(q)$ (if we were to calculate average computation times, we would be interested in the average size). Observe that for all nodes q it holds that q is inserted into W only when something is added to $\mathsf{D}(q)$, i.e., at most H times; therefore each "clause" in the body of `iter` (cf. the sample residual program in Fig. 2) is called at most H times. This demonstrates that $R \in O(HS) \subseteq O(H(N + AL))$, implying $R \in O(N^3)$ (since $H \leq N$, trivially).

[2] Note that instead of generating a function $\texttt{iter}_q$ for each q, it generates (so as to optimise performance) a single function `iter` which then selects the appropriate code by binary search.

```
fun RCFA1idid () = let
 val W = ref ([]:int list)
 val D = ArrI (8,Data.empty)
 fun addD q d = ... (* as in interpreter *)
 val _ = (addD 6 (Data.mk_singleton 6) ; addD 3 (Data.mk_singleton 3) )

 fun iter n =
  if n < 5 then
   if n < 3 then
    if n < 2 then iterate ()
    else (addD 4 (ArrL (D,2)) ; iterate () )
   else if n < 4 then
     ( if Data.is_member (3,ArrL(D,3)) then addD 2 (ArrL(D,6)) else ()
     ; if Data.is_member (3,ArrL(D,3)) then addD 1 (ArrL(D,4)) else ()
     ; if Data.is_member (6,ArrL(D,3)) then addD 5 (ArrL(D,6)) else ()
     ; if Data.is_member (6,ArrL(D,3)) then addD 1 (ArrL(D,7)) else ()
     ; iterate ())
    else
     ( if Data.is_member (3,ArrL(D,3)) then addD 1 (ArrL(D,4)) else ()
     ; iterate ())
  else if n < 7 then
    if n < 6 then (addD 7 (ArrL(D,5)) ; iterate () )
    else
     ( if Data.is_member (3,ArrL(D,3)) then addD 2 (ArrL(D,6)) else ()
     ; if Data.is_member (6,ArrL(D,3)) then addD 5 (ArrL(D,6)) else ()
     ; iterate ())
   else
    ( if Data.is_member (6,ArrL(D,3)) then addD 1 (ArrL(D,7)) else ()
    ; iterate ())
 and iterate () = case !W of [] => ()
                  | (q::W') => (W := W'; iter q)
 val _ = iterate ()
in fn q => (print ((Data.print (ArrL (D,q)))^"\n"); ())  end
```

Figure 2: The constraint-solving approach: the residual code for *idid*

By similar considerations (observing that the time spent by `process_ccs` is bounded by the number of constraints and therefore in $O(N + AL)$), we infer that $I \in O(H(N + AL))$ also.

We have thus provided cubic time bounds for our closure analyses. This is in fact the well-known complexity of 0CFA "with inclusions". (For CFA "with equalities", a technique introduced by Henglein [8], the complexity is almost linear but at the price of slightly less precise results.)

3 CFA by solving constraints: an incremental approach

In Sect. 2.2 we estimated the size of the residual program to be in $O(N + AL)$, indicating that the approach taken so far does not scale up well. In fact, looking at Fig. 2 should convince the reader that the residual code in general mostly consists of tests where the condition takes the form `Data.is_member` $(l,$`ArrL` $($`D`$,q))$. Worse yet, for typical programs only a few lambda abstractions will reach any given function application (i.e., H will be much smaller than L), and hence the large majority of these tests will always be false.

This unpleasant property of the residual programs clearly reflects that also the *interpreter* spends most of its time on such "superfluous" tests. This (not too deep) observation inspired us to (re)invent a better algorithm. The idea is to avoid the explicit generation of conditional constraints, and instead to proceed as follows: whenever `addD` adds a new element $(\mathtt{lambda}\ (\mathtt{l}'_1\ \ldots\ \mathtt{l}'_n)\ \mathtt{l}')^{l'_0}$ to $\mathsf{D}(l_0)$, where l_0 is the operand part of an application $(\mathtt{l}_0\,\mathtt{l}_1\ \ldots\ \mathtt{l}_n)^l$, it calls the function `add_newcls` on l_0 and $\{l'_0\}$ so as to generate the *unconditional* constraints $\mathsf{D}(l_1) \subseteq \mathsf{D}(l'_1)$, ..., $\mathsf{D}(l_n) \subseteq \mathsf{D}(l'_n)$, $\mathsf{D}(l') \subseteq \mathsf{D}(l)$. To represent such constraints, we use an array E_d which can be viewed as the dynamic part of the E from Fig. 1. Then we for $i \in \{1 \ldots n\}$ add l'_i to $\mathsf{E}_d(l_i)$, and add l to $\mathsf{E}_d(l')$.

An algorithm implementing this idea is depicted in Fig. 3. It takes as input a set of unconditional constraints (from which it constructs E_s, the static counterpart of E_d), together with functions `cl2inf` and `fnp2ra`. The former associates each lambda abstraction with its body and formal parameters; the latter associates each operand with its call site and its actual parameters.

It turns out that we have essentially reinvented a well-known technique, described, e.g., by Mossin [10, Sect. 4.1] and by Wright and Jagannathan (for a polyvariant analysis) [15, Fig. 6]. We believe that this illustrates a major virtue of the partial-evaluation approach: no matter how skillful one is, there exists problems where the design of efficient algorithms may be greatly assisted by looking at their residual code, since there the flow of control is more explicit so that one can pinpoint sources of inefficiency.

Also for this interpreter, a generating extension can be written: for our example program, this results in the residual program depicted in Fig. 4. We observe that for `iter`, only a small amount of interpretation overhead has been eliminated; this is because no constraints except those in E_s ($\mathsf{D}(2) \subseteq \mathsf{D}(4)$ and $\mathsf{D}(5) \subseteq \mathsf{D}(7)$) are available before `iter` is actually run.

3.1 Complexity estimates

For the algorithm presented in this section, we now redo the complexity analysis from Sect. 2.2. As there are $O(N)$ unconditional constraints it is easy to see that $S \in O(N)$, i.e., the size of the residual code is linear in the size of the source program. This is of course essential for all practical purposes!

Concerning I, first note that $O(AH)$ elements are placed in E_d. Observe that the total time spent in a call to `addD` is bounded by a function $c_1 + c_2U$, where U is the number of constraints inserted in E_d during the call (including recursive invocations). So if we add $O(AH)$ to the final time bound we can assume that each call to `addD` takes time $O(1)$. As in the previous analysis, we see that each node q is inserted into W at most H times so for each q it holds that `iter` and `iter_d` is each called at most H times on q. The cost of applying `iter` to q is bounded by the size of $\mathsf{E}_s(q)$, and the cost of applying

```
fun CFA2 (size,lccs,(cl2inf,fnp2ra)) = let
 val Es = ArrI (size,[]:int list)
 fun addEs q q' = ArrU (Es,q, q'::(ArrL (Es,q)))
 val W = ref ([]:int list)
 val Ed = ArrI (size,[]:int list)
 val D = ArrI (size,Data.empty)
 fun addD q d = let val Dq = ArrL (D,q)
                   val newcls = Data.set_diff (d,Dq)
                in if Data.is_empty newcls then ()
                   else ( ArrU (D,q, Data.union (newcls,Dq))
                        ; W := (q::(!W))
                        ; add_newcls q newcls) end
 and add_newcls q newcls = ...
        (* updates Ed (using fnp2ra and cl2inf)   *)
        (* for each new constraint q1 <= q2
             it calls addD q2 (ArrL(D,q1))        *)
 fun process_lccs [] = ()
 |   process_lccs ((NodeIncl (q1,q2)) :: lccs) =
               (addEs q1 q2; process_lccs lccs)
 |   process_lccs ((DataIncl (l,p)) :: lccs) =
               (addD p (Data.mk_singleton l); process_lccs lccs)
 val _ = process_lccs lccs

 fun iter q = let fun iter_s [] = ()
                 |   iter_s (q1::qs) = ( addD q1 (ArrL (D,q))
                                       ; iter_s qs)
               in iter_s (ArrL (Es,q)) end
 and iter_d q [] = iterate ()
 |   iter_d q (q1::qs) = (addD q1 (ArrL (D,q)); iter_d q qs)
 and iterate () = case !W of
          [] => ()
        | (q::W') => ( W := W'; iter q; iter_d q (ArrL (Ed,q)))
 val _ = iterate ()
in fn q => (print ((Data.print (ArrL (D,q)))^"\n"); ())  end
```

Figure 3: The lazy constraint-solving approach

`iter_d` to q is bounded by the size of $\mathsf{E}_d(q)$ (as we could assume $\texttt{addD} \in O(1)$). This shows that $I \in O(H(N + AH))$. By similar considerations, we infer that $R \in O(H(N + AH))$ also. This is still in $O(N^3)$, but recall that in Sect. 2.2 we were only able to establish the bound $R = I \in O(H(N + AL))$. This indicates that in practice, where H is far less than L, the algorithm presented in this section is a substantial improvement over the one from Sect. 2.

4 Benchmarks

Using Standard ML of New Jersey[3] [2], we have implemented the two versions of CFA that were presented in Sects. 2 and 3. Our system is able to handle

[3] Version 110.0.3

```
fun RCFA2idid () = let
 fun cl2inf_idid 3 = SOME (BodyFormals (4,[2]))
 |   cl2inf_idid 6 = SOME (BodyFormals (7,[5]))
 |   cl2inf_idid n = NONE
 fun fnp2ra_idid 3 = SOME (1,[6])
 |   fnp2ra_idid n = NONE
 val W = ref ([]:int list)
 val Ed = ArrI (8,[]:int list)
 val D = ArrI (8,Data.empty)
 fun addD q d = ...           (* as in interpreter *)
 and add_newcls q newcls = ... (* as in interpreter *)
 val _ = ( addD 6 (Data.mk_singleton 6)
         ; addD 3 (Data.mk_singleton 3) )

 fun iter n =
  if n < 5 then
   if n < 3 then
    if n < 2 then ()
    else (addD 4 (ArrL (D,2)) ; () )             (* n = 2 *)
   else ()
  else if n < 6 then (addD 7 (ArrL (D,5)) ; () ) (* n = 5 *)
   else ()
 and iter_d q [] = iterate ()
 |   iter_d q (q1::qs) = (addD q1 (ArrL (D,q)); iter_d q qs)
 and iterate () =
          case !W of [] => ()
              | (q::W') => (W := W'; iter q ; iter_d q (ArrL (Ed,q)))
 val _ = iterate ()
in fn q => (print ((Data.print (ArrL (D,q)))^"\n"); ())  end
```

Figure 4: The lazy constraint-solving approach: the residual code for *idid*

source programs written in a large subset of the Scheme language: by means of a front end[4] such programs are parsed, labeled, and translated into a reduced Scheme subset for which constraint generation is straightforward[5].

We have considered Feeley's Scheme benchmarks [3]:

conform A program that manipulates lattices and partial orders

earley A parser generator based on Earley's algorithm

interp An environment-based call-by-need λ-calculus interpreter

lambda A substitution-based call-by-name λ-calculus interpreter

lex A generator of lexical analyzers

[4]Written by Daniel Damian.

[5]The treatment of primitive operators is somewhat crude: for an application $(\mathtt{l}_0\ \mathtt{l}_1\ \ldots\ \mathtt{l}_n)^l$ we stipulate that if l_0 may evaluate to a primitive operator then for each $i \in \{1 \ldots n\}$ it must hold that $\mathsf{D}(l_i) \subseteq \mathsf{D}(l)$. For this to be sound, we must demand that the user has priorly eliminated all occurrences of higher-order primitive operators such as `apply`.

	N	A	L	AL	I(ms)	G(sec)	S(kb)	R(ms)	speedup
idid	8	1	2	2	.039	.0024	1.35	.023	40 %
S'id3	26	6	6	36	.41	.044	12.7	.35	15 %
factest	243	60	12	720	2.77	.256	121	1.38	50 %
conform	2234	476	103	49,028	306	26.9	7,737		
earley	2748	489	79	38,631	311	18.5	5,057		
interp	1422	308	84	25,872	238				
lambda	2952	759	57	43,263	280				
lex	4628	1041	126	131,166	715				
ll1	2152	490	99	48,510	394				
peval	2764	688	76	52,288	519				
source	1566	339	57	19,323	142				

Table 1: Experiments with the algorithm of Fig. 1

ll1 An $LL(1)$ parser generator

peval A small partial evaluator for Scheme

source A parser for Scheme

Apart from these we have also included some smaller benchmarks: our running example *idid*, a program *S'id3* that applies the combinator $\lambda f.\lambda y.\lambda x.((fy)(fx))$ to three copies of the identity function, and a program *factest* that calls the factorial function in various ways.

When running our programs on the benchmarks, we have measured I, R and S as defined in Sect. 2.2; additionally we have included G, the time used by the generating extension to produce the residual code. (When interpreting the figures, keep in mind that we are only interested in *relative* numbers; as to be expected, G turns out to be roughly proportional to S.) We do not count the time spent on compiling the residual program, since we can imagine that the generating extension would output machine code directly.

For the original algorithm (Fig. 1), the result of running selected benchmarks is depicted in Table 1. It turns out that only the small benchmarks give rise to residual programs with a manageable (compilable) size; in these cases the speedup factor (the relationship between R and I) seems to be slightly less than 2.

We might also test our estimates from Sect. 2.2, where we predicted that $S \in O(N + AL)$ and that $I \in O(H(N + AL))$. For the Feeley benchmarks N is much less than AL, and apparently H is a small constant (often even 1). Therefore we may expect I and S to be proportional to AL, and this seems to be pretty much the case.

For the improved algorithm from Sect. 3, the result of running the benchmarks is depicted in Table 2. Observe that now all residual programs are small and thus can be easily compiled.

Again, we might like to test our complexity estimates, this time from Sect. 3.1. Here we predicted that $S \in O(N)$ and that $I, R \in O(H(N + AH))$. Assuming

	N	A	L	AL	I(ms)	G(ms)	S(kb)	R(ms)	speedup
idid	8	1	2	2	.042	1.21	1.4	.034	20 %
S'id3	26	6	6	36	.24	3.0	2.0	.21	12 %
factest	243	60	12	720	1.16	35.8	9.7	1.08	7 %
conform	2234	476	103	49,028	15.6	270	91	14.0	10 %
earley	2748	489	79	38,631	41.9	412	127	44.1	−5 %
interp	1422	308	84	25,872	28.7	165	58	27.7	3 %
lambda	2952	759	57	43,263	21.1	302	105	19.3	8 %
lex	4628	1041	126	131,166	34.6	690	208	32.1	7 %
ll1	2152	490	99	48,510	14.3	256	89	12.9	10 %
peval	2764	688	76	52,288	20.6	306	107	18.0	12 %
source	1566	339	57	19,323	8.9	193	67	8.8	1 %

Table 2: Experiments with the algorithm of Fig. 3

that H is so small that AH is less than N, this means that I, R and S all are in $O(N)$. And indeed, this seems pretty much to be the case.

Concerning the relationship between I and R, observe that we typically experience the expected minor speedup. For a single benchmark, however, we experience a slowdown. This might not have been the case if the generating extension did output machine code directly, rather than relying on a compiler which is general purpose (able to deal with the full language SML and tuned for typical handwritten programs) and therefore by necessity not too efficient on automatically generated, low-level specialized programs.

5 Conclusion

Partial evaluation is generally applicable for program analysis since all program analyses share the common property of traversing a source program (or some representation derived from it) repeatedly whenever they have to compute fixed points. On the other hand, partial evaluation only contributes to eliminating the interpretive overhead of program analysis; the other dynamic operations remain—such as set operations in CFA. At any rate, partial evaluation provides a linear improvement: it does not change the complexity of the analysis.

The benefits of partial evaluation are both conceptual and practical. Conceptual: residual programs often provide a better comprehension of the source analysis, perhaps enabling one to optimize it, as illustrated by this case study. And practical: the resulting analysis is more efficient, assuming that partial evaluation and compiling the residual programs do not take too long. It has been our consistent observation that residual programs in our case are very low level and regular, thus making a strong case for runtime code generation—a future work.

For the CFA in Sect. 3, which essentially uses dynamic programming, there is little benefit in applying static partial evaluation. One might rather perform dynamic partial evaluation, i.e., incremental runtime code generation.

6 Acknowledgments

This work grew out of a joint project with Daniel Damian and Olivier Danvy, initiated while the author was at BRICS (University of Aarhus), on investigating the use of partial evaluation for various kinds of program analysis[6]. I would like to thank Olivier for proposing the project, and to thank both Daniel and Olivier for valuable feedback at all stages. All of us are grateful to Dominique Boucher for sending us Marc Feeley's Scheme benchmarks.

References

[1] Torben Amtoft. Properties of unfolding-based meta-level systems. In *Proceedings of Partial Evaluation and Semantics-Based Program Manipulation (PEPM '91)*, Sigplan Notices 26(9), pages 243–254, New Haven, Connecticut, June 1991.

[2] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[3] Dominique Boucher and Marc Feeley. Abstract compilation: a new implementation paradigm for static analysis. In T. Gyimothy, editor, *Proceedings of CC'96, the 6th International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, Linköping, Sweden, April 1996. Springer-Verlag.

[4] Charles Consel and Olivier Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, Vol. 30, No. 2, pages 79–86, January 1989.

[5] 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.

[6] Daniel Damian. Partial evaluation for program analysis. Progress report, BRICS PhD School, University of Aarhus, June 1999.

[7] Kirsten L. Solberg Gasser, Flemming Nielson, and Hanne Riis Nielson. Systematic realisation of control flow analyses for CML. In Mads Tofte, editor, *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 38–51, Amsterdam, The Netherlands, June 1997. ACM Press.

[6] Another spinoff of this project is Daniel's progress report [6], treating Shivers' 0CFA [14].

[8] Fritz Henglein. Simple closure analysis. Technical Report Semantics Report D-193, DIKU, Computer Science Department, University of Copenhagen, 1992.

[9] 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.

[10] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, Computer Science Department, University of Copenhagen, 1997. Technical Report DIKU-TR-97/1.

[11] Flemming Nielson and Hanne Riis Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In Neil D. Jones, editor, *Proceedings of the Twenty-Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 332–345, Paris, France, January 1997. ACM Press.

[12] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.

[13] Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91, the ACM SIGPLAN Sixth Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 146–161, Phoenix, Arizona, October 1991.

[14] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

[15] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: an effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems*, Vol. 20, No. 1, pages 166–207, January 1998.

Recent BRICS Report Series Publications

RS-99-45 **Torben Amtoft.** ***Partial Evaluation for Constraint-Based Program Analyses*****. December 1999. 13 pp.**

RS-99-44 **Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro.** ***Aliasing Models for Mobile Objects*****. December 1999. ii+46 pp. To appear in a special FOOL6 issue of *Information and Computation*. An extended abstract of this revision, entitled *Aliasing Models for Object Migration*, appeared as Distinguished Paper in Amestoy, Berger, Daydé, Duff, Frayssé, Giraud and Daniel, editors, *5th International Euro-Par Conference*, EURO-PAR '99 Proceedings, LNCS 1685, 1999, pages 1353–1368, which in turn is a revised part of another paper called *Migration = Cloning ; Aliasing* that appeared in Cardelli, editor, *Foundations of Object-Oriented: 6th International Conference*, FOOL6 Informal Proceedings, 1999 and as such supersedes the corresponding part of the earlier BRICS report RS-98-33.**

RS-99-43 **Uwe Nestmann.** ***What is a 'Good' Encoding of Guarded Choice?*** **December 1999. ii+34 pp. To appear in a special EXPRESS '97 issue of *Information and Computation*. This revised report supersedes the earlier BRICS report RS-97-45.**

RS-99-42 **Uwe Nestmann and Benjamin C. Pierce.** ***Decoding Choice Encodings*****. December 1999. ii+62 pp. To appear in *Journal of Information and Computation*. An extended abstract appeared in Montanari and Sassone, editors, *Concurrency Theory: 7th International Conference*, CONCUR '96 Proceedings, LNCS 1119, 1996, pages 179–194.**

RS-99-41 **Nicky O. Bodentien, Jacob Vestergaard, Jakob Friis, Kåre J. Kristoffersen, and Kim G. Larsen.** ***Verification of State/Event Systems by Quotienting*****. December 1999. 17 pp. Presented at *Nordic Workshop in Programming Theory*, Uppsala, Sweden, October 6–8, 1999.**

RS-99-40 **Bernd Grobauer and Zhe Yang.** ***The Second Futamura Projection for Type-Directed Partial Evaluation*****. November 1999. Extended version of an article appearing in Lawall, editor, *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '00 Proceedings, 2000, pages 22–32.**