Combining Abstract Interpretation and Partial Evaluation (brief overview)

Neil D. Jones*

DIKU, University of Copenhagen
e-mail: neil@diku.dk

Abstract. Earlier work has shown that partial evaluation of a self-interpreter with respect to an input program can be used to effect program transformation. This talk shows that this technique, augmented by a preliminary abstract interpretation, can yield, among many other transformations:

- Safe, efficient execution of programs in an unsafe language, and
- A change in program style, from one list representation to another

Further, partial evaluation can be viewed as an on-line polyvariant abstract interpretation, augmented by code generation. This view generalizes conventional projection-based partial evaluation in the direction of Turchin's supercompilation.

Partial evaluation for optimization and other program transformations

Beginning programmers, once they discover that the same operation may be performed in various ways of differing efficiency, sometimes write code such as "if $x = 2$, then compute $x * y$ by a left shift, else use the multiply instruction." This is, of course, useless: The time it takes to determine whether or not $x = 2$ exceeds the time saved by performing a shift instead of a multiply.

The situation can be quite different, however, if a partial evaluator is used to transform a source program `pgm` by specializing a self-interpreter `int`. By definition of partial evaluation $[\![\mathtt{q}]\!](\mathtt{s},\mathtt{d}) = [\![\mathtt{q}_\mathtt{s}(\mathtt{d})]\!] = [\![[\![\mathtt{mix}]\!](\mathtt{q},\mathtt{s})]\!](\mathtt{d})$ for any program `q` with inputs `s` and `d`.

Recall that a self-interpreter should satisfy, for any program `p` and input `d` (notation from [7]):

$$[\![\mathtt{p}]\!](\mathtt{d}) = [\![\mathtt{int}]\!](\mathtt{p},\mathtt{d})$$

Partial evaluation can compute a specialized program by specializing the self-interpreter:

$$\mathtt{p}' = \mathtt{int}_\mathtt{p} = [\![\mathtt{mix}]\!](\mathtt{int},\mathtt{p})$$

* This research was partially supported by the Danish *DART* project.

Thus the transformation of p into $\mathtt{p}' = \mathtt{int}_{\mathtt{p}}$ yields a semantically equivalent program, that is $[\![\mathtt{p}']\!] = [\![\mathtt{p}]\!]$. The transformed program $\mathtt{p}'$ may be more or less efficient than p. If mix is "optimal" (as defined in [7]) then $\mathtt{p}'$ will always be at least as fast as p.

An interpreter can exploit the results of abstract interpretation

The naive reasoning of the beginner can be of some use if it can be determined *before program execution begins* that $x = 2$. The point is that information about the program being executed which can be discovered statically (from the program alone, without its input data), can be exploited by the interpreter to select among alternate ways to do a given calculation. Since the information is static, tests on it will be removed during specialization.

We now make the technique more explicit. The first step is to devise an appropriate framework for abstract interpretation (static program analysis, see [3, 1, 8]), whose abstract values are relevant to deciding when the interpreter may use more efficient operations. Let $\mathtt{p}^{ann}$ denote the result of an abstract interpretation. This consists of the original program p, *annotated* with descriptions of the values manipulated at the various program points in p, where the descriptions are given by values over an abstract interpretation lattice or other domain.

The second step is to write an interpreter for the language which has as input:

1. The program to be executed,
2. Abstract values describing the values of its variables at every program point, and
3. The program's own input.

The interpreter is written to execute the program correctly; but also to exploit the abstract values wherever possible to improve execution speed:

$$[\![\mathtt{p}]\!](\mathtt{d}) = [\![\mathtt{int}]\!](\mathtt{p}^{ann}, \mathtt{d})$$

Of course, the savings realized by using fast operations are outweighed by the costs of testing the annotations, and these are again outweighed by the costs of running interpretively. The situation is quite different, however, when partial evaluation is applied since neither the interpretive overhead nor the abstract interpretation occur at run-time.

Program transformation from p to $\mathtt{p}'$ is thereby done in two steps:

- Perform the abstract interpretation to obtain $\mathtt{p}^{ann}$
- Specialize the self-interpreter, using the program and the abstract values as static input:

$$\mathtt{p}' = \mathtt{int}_{\mathtt{p}^{ann}} = [\![\mathtt{mix}]\!](\mathtt{int}, \mathtt{p}^{ann})$$

Safe execution of programs in an unsafe language

Assume given a language in which operations may be performed in two ways: "fast but unsafe" and "safe but slow". Combination of abstract interpretation and partial evaluation can yield a transformer *from* a program written using only "fast" operations *to* an equivalent version using "safe but slow" operations only as necessary to avoid run-time errors.

For example, a store operation into a memory address (direct or computed) might require the address both to lie within the current program's data address space, and as well to be a multiple of 4. For this example it would be natural to abstract each variable's values by two things: its value modulo 4, using a lattice with elements $\{\bot, 0, 1, 2, 3, \top\}$; and an interval such as $[0, 32767]$ or $[1024, \infty]$.

Construct a self-interpreter `int` that consults the abstract value whenever it is to perform an operation to store into location L. Suppose the program's legal address area is the interval $[\ell, u]$, and that the abstract value for this address expression is $m \in \{\bot, 0, 1, 2, 3, \top\}$ together with an interval $[a, b]$. The interpreter's actions:

- If $m \in \{\bot, 1, 2, 3\}$ or $[a, b] \cap [\ell, u] = \{\}$, then issue an error message and stop immediately.
- If $m = 0$ and $[a, b] \subseteq [\ell, u]$, then perform the store operation without testing.
- Otherwise: if $m = \top$ then perform a test $L \bmod 4 = 0?$; and if $[a, b] \setminus [\ell, u] \neq \{\}$ then perform a range test $L \in [\ell, u]?$. Stop if either answer is "no," else perform the store operation.

Changing programs from one abstract data type into another

A related application of this technique is to "list compression": translating a program using conventional Lisp-like "1-lists" with one data field and one pointer per cell, into an equivalent version that uses a more compact "2-list" form with 2 data fields per cell (and one pointer). For large n, with the 2-list representation 50% of the memory to store an n-element list consists of pointers. This proportion is reduced to 33% when using the 2-list representation. Following are ML-like datatype declarations for 1-lists and 2-lists:

```
datatype list  = NIL | CONS of int * list

datatype list2 = OD of int * tail2
               | EV of tail2

     and tail2 = NL2
               | CN2 of int * int * tail2
```

Sequence representation depends on its *parity*. i.e. whether its length is even or odd:

$[a_1, a_2, \ldots, a_{2n}] \quad \Rightarrow \texttt{EV}(\texttt{CN2}(a_1, a_2, \texttt{CN2}(a_3, a_4, \ldots, \texttt{CN2}(a_{2n-1}, a_{2n}, \texttt{NL2})\ldots)))$
$[a_1, a_2, \ldots, a_{2n+1}] \Rightarrow \texttt{OD}(a_1, \texttt{CN2}(a_2, a_3, \ldots, \texttt{CN2}(a_{2n}, a_{2n+1}, \texttt{NL2})\ldots))$

A natural first step is to program an interpreter that accepts a program manipulating 1-lists, but simulates it assuming that input and output data are all represented in 2-list form. This is straightforward, except that some code is needed to keep track, or determine, the parity of an entry in a 2-list. For instance, a "cons" operation would require a test:

```
fun CNS x EV(ys)     =  OD(x,ys))
fun CNS x OD(y, ys)  =  EV(CN2(x,y,ys))
```

Consider the "append" program:

```
fun ap nil  ys        =  ys
fun ap cons(x,xs) ys  =  cons(x, (ap xs ys))
```

Specializing the interpreter just described with respect to the 1-list program "append" indeed gives an equivalent 2-list program. However, its efficiency is less than desirable: it actually runs slower than the original. The reason is due to multitudinous tests carried out on the parity of the arguments.

A natural question is whether this can be improved? For instance, it seems natural that "append" should only need to check the parity of its two arguments at the start of execution, whereafter it should be able to proceed straight ahead down the data lists, performing one CN2 operation every time it sees two new data values.

The answer is "yes." The technique is to perform an abstract interpretation on the results yielded by "append." A polyvariant parity analysis reveals that "append" takes (even,even) and (odd, odd) to even, and the two mixed argument combinations to odd. Equipping the interpreter to exploit this information, code equivalent to the following will be obtained:

```
fun ap EV(xs)           EV(ys)     =  EV(evens xs ys)
     | OD(x, xs)        EV(ys)     =  OD(x, evens xs ys)
     | OD(x, xs)        OD(y, ys)  =  EV(odds x xs y ys)
     | EV(NL2)          OD(y, ys)  =  OD(y, ys)
     | EV(CN2(u,v,xs))  OD(y, ys)  =  OD(u,odds v xs y ys))

fun evens NL2 ys                    =  ys
      |  CN2(u,v,xs) ys             =  CN2(u,v,(evens xs ys))

fun odds x NL2         y ys         =  CN2(x,y,ys)
      |  x CN2(u,v,xs) y ys         =  CONS2(x,u,(odds v xs y ys))
```

Such a program transformation has been studied before, using entirely different methods: *refinement types* in [13], and second-order polymorphic lambda calculus in [6]. Our point here is that such transformation can be performed

by specializing an interpreter; and that more efficient transformed code can be obtained by a preliminary abstract interpretation.

Experiments. Saumya Debray has implemented this transformation and tried it out on a number of programs. It seems to work well when the lengths of the lists involved vary systematically (e.g. append and merge sort), and less well for quicksort: The unpredictability of the sublist lengths causes the transformed program to contain many tests, reducing its efficiency. Further, some experiemnts revealed unexpected interactions with other compiler optimizations, for example to exploit tail recursion, and special treatment some compilers give to binary lists.

Partial evaluation as on-line abstract interpretation

It is known that Turchin's "supercompilation" methodology [14] for program transformation is more powerful than partial evaluation as usually practiced. For example, Turchin said in 1987 that supercompilation could specialize a naive pattern-matching algorithm so as to obtain the efficient linear-time matcher output by the Knuth-Morris-Pratt algorithm [12].

Since this was well beyond the reach of current partial evaluators, it was natural to ask the question: Why is it possible? At first some thought it was due to the programming language Refal used by the first supercompilers, with associative nested symbol strings as data structures. This was shown false, however, by the results of [5] using a rather different language.

Paper [11] shows that a generalization of the usual approaches to partial evaluation can achieve this effect, and so captures at least a part of supercompilation. This approach is seen to give specializations not achieved by the traditional framework. Examples include use of parity information, and relational abstract interpretations to obtain the Knuth-Morris-Pratt result by specializing a naive string matcher.

The main ideas are the following.

Generalizing partial evaluation towards supercompilation. Many existing partial evaluation algorithms use as a partially known value a *store projection*: A partial-evaluation-time representation of a run-time state consists of the completely known values of some variables, and "unknown" for others.

A more general approach to manipulate partial evaluation-time values is to use an abstract value that describes a *set of stores*, rather than just the known parts of one particular store. Using a flowchart framework for simplicity, the idea is to simulate a computation

$$(p_0, s_0) \rightarrow (p_1, s_1) \rightarrow (p_2, s_2) \rightarrow \ldots$$

by means of an *instrumented computation*

$$((p_0, \overline{s}_0), s_0) \rightarrow ((p_1, \overline{s}_1), s_1) \rightarrow ((p_2, \overline{s}_2), s_2) \rightarrow \ldots$$

Here $p_0, p_1, \ldots$ are *program points*, e.g. labels in the program being executed, $s_0, s_1, \ldots$ are *stores*, i.e. mappings of variables to values. At specialization time the partial evaluator manipulates *sets of stores* $\overline{s}_0, \overline{s}_1, \ldots$. An invariant will be maintained: that $s_i \in \overline{s}_i$ for all i, i.e. that the store sets "cover" any actual stores entered into by the program.

The information given by this invariant is *the source of all improvements* gained by this type of program specialization. Its use is to optimize the specialized program by generating equivalent but more efficient code exploiting the information given by $\overline{s}$. In particular some computations may be elided altogether, since their effect can be achieved by using the $\overline{s}$ at transformation time. Further, knowledge of $\overline{s}$ often allows a very economical representation of the stores $s \in \overline{s}$, since their common features are known at specialization time.

Relation to abstract interpretation. During program transformation the partial evaluator does not, of course, manipulate actually infinite sets of stores, but rather only finite descriptions of them. Further, the partial evaluator does not simulate all possible computations, but rather collects together enough store set descriptions to cover all states the program can reach in any run-time computation. The results of this are used as a basis for generating residual code to appear in the specialized program (not emphasized in this overview).

The original Cousot and Cousot framework [3] involving "abstraction" and "concretization" functions, upper closure operators, etc. is perfectly well-suited to describe this process. From this viewpoint, the projections used in much partial evaluation are thus an "independent attribute" abstraction, whereas more general descriptions of store sets form a "relational" abstraction of a store set (terminology from [9]).

An example: Collatz' problem in number theory. This problem amounts to determining whether the following program terminates for all positive n. To our knowledge it is still unsolved. We will show how an abstract imterpretation can be used to transform it into another version which is somewhat faster.

```
A:  if n = 1 goto G;
   B:  if n even
       then (C: n := n ÷ 2;)
       else (D: n := 3 * n + 1;)
       fi;
    goto A;
G:
```

The program has only one variable n, so a store set is essentially a set of values. We use just four store sets:

$$
\begin{aligned}
\mathit{Even} &= \{[n \mapsto x] \mid x \in \{0, 2, 4, \ldots\}\} \\
\mathit{Odd} &= \{[n \mapsto x] \mid x \in \{1, 3, 5, \ldots\}\} \\
\top &= \{[n \mapsto x] \mid x \in \mathcal{N}\} \\
\bot &= \{\}
\end{aligned}
$$

A polyvariant abstract interpretation gives the following reachable store sets.

$$\{(\mathtt{A}, \top), (\mathtt{G}, \mathit{Odd}), (\mathtt{B}, \top), (\mathtt{C}, \mathit{Even}), (\mathtt{D}, \mathit{Odd}), (\mathtt{A}, \mathit{Even}), (\mathtt{G}, \bot), (\mathtt{B}, \mathit{Even}), (\mathtt{D}, \bot)\}$$

Assembling these program points into a new program we get a new program.

```
A_⊤:  if n = 1 goto G_Odd;
B_⊤:  if n even
      then C_Even: n := n ÷ 2;
      else D_Odd: n := 3 * n+1;
           A_Even: if n = 1 goto B_Even else G_⊥;
           B_Even: if n even then goto C_Even
                                else goto D_⊥
      fi;
   goto A_⊤;
G_Odd:
```

The labels $\mathtt{D}_\bot$, $\mathtt{G}_\bot$ are inaccessible (empty sets of stores!), so the tests leading to them may be elided. Removing them gives the following somewhat faster program:

```
A_⊤:  if n = 1 goto G_Odd;
B_⊤:  if n even
      then C_Even: n := n ÷ 2;
      else D_Odd: n := 3 * n+1;
           goto C_Even
      fi;
   goto A_⊤;
G_Odd:
```

The example is interesting not for what it does, but because it has a different flavor than the usual partial evaluation, due to the use of abstract interpretation.

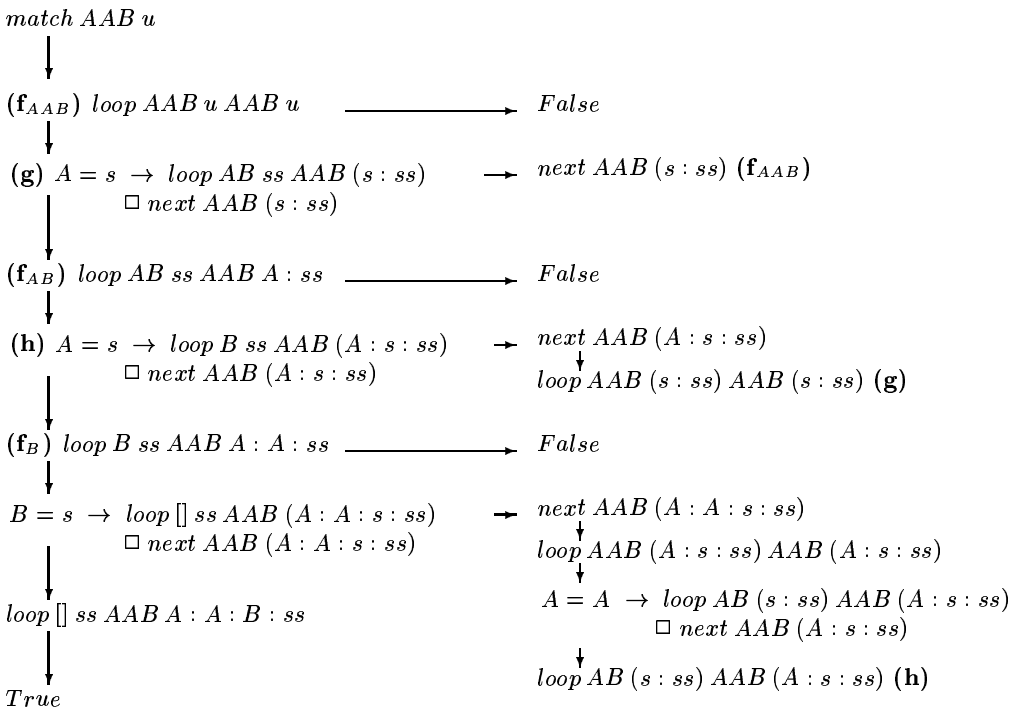
An example: string pattern matching. In this example we use as a state description a term containing constants (the symbols A, B) and built up using concatenation (:) from program configurations. For example, the term

loop AB ss AAB $(s : ss)$ describes the set of all states obtained by replacing the ss by one value, and s by another value (the same in both occurrences).

Following is a simple program for string pattern matching; it determines whether pattern p occurs as a contiguous substring of subject s.

$$
\begin{array}{ll}
match\ p\ s & = loop\ p\ s\ p\ s \\
 & \\
loop\ []\ ss\ op\ os & = True \\
loop\ (p : pp)\ []\ op\ os & = False \\
loop\ (p : pp)\ (s : ss)\ op\ os & = \textbf{if}\ p = s\ \textbf{then}\ loop\ pp\ ss\ op\ os\ \textbf{else}\ next\ op\ os \\
 & \\
next\ op\ [] & = False \\
next\ op\ (s : ss) & = loop\ op\ ss\ op\ ss
\end{array}
$$

The following diagram contains an exhaustive set of state descriptions for this program, if run with fixed pattern AAB and an unknown subject string s.



This set of states consists largely of known constants, and so can be converted

to a significantly simpler program.

$$
\begin{array}{ll}
f\ u & = f_{AAB}\ u \\
 & \\
f_{AAB}\ [] & = False \\
f_{AAB}\ (s : ss) & = g\ s\ ss \\
 & \\
g\ s\ ss & = \textbf{if}\ A = s\ \textbf{then}\ f_{AB}\ ss\ \textbf{else}\ f_{AAB}\ ss \\
 & \\
f_{AB}\ [] & = False \\
f_{AB}\ (s : ss) & = h\ s\ ss \\
 & \\
h\ s\ ss & = \textbf{if}\ A = s\ \textbf{then}\ f_{B}\ ss\ \textbf{else}\ g\ ss \\
 & \\
f_{B}\ [] & = False \\
f_{B}\ (s : ss) & = \textbf{if}\ A = s\ \textbf{then}\ g\ s\ ss\ \textbf{else} \\
 & \quad \textbf{if}\ B = s\ \textbf{then}\ true\ \textbf{else}\ h\ s\ ss
\end{array}
$$

Note that the transformed program scans through the subject string only once; the repeated backing up seen in the original program has vanished.

References

1. Samson Abramsky and Chris Hankin, editors. *Abstract Interpretation of Declarative Languages.* Ellis Horwood, 1987.
2. Charles Consel and Olivier Danvy, Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30, pp. 79–86, January 1989.
3. Patrick Cousot and Radhia Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Fourth ACM Symposium on Principles on Programming Languages*, pp. 238–252, New York: ACM Press, 1977.
4. Andrei P. Ershov. Mixed computation: Potential applications and problems for study. *Theoretical Computer Science*, 18, pp. 41–67, 1982.
5. Robert Glück and Andrei V. Klimov, Occam's razor in metacomputation: the notion of a perfect process tree. In *Static analysis Proceedings*, eds. P. Cousot, M. Falaschi, G. Filé, G. Rauzy. Lecture Notes in Computer Science 724, pp. 112-123, Springer-Verlag, 1993.
6. C.V. Hall, Using Hindley-Milner type inference to optimise list representation. In *ACM Conference on Lisp and Functional Programming*, pp. 162–172, ACM Press, 1994.
7. Neil D. Jones, C.K. Gomard, P. Sestoft, *Partial Evaluation and Automatic Program Generation*, Prentice Hall International Series in Computer Science, 1993.
8. Neil D. Jones and Flemming Nielson, Abstract interpretation: a semantics-based tool for program analysis, 122 pages. In *Handbook of Logic in Computer Science*, Oxford University Press, 1995.

9. Neil D Jones and Steven S Muchnick, Complexity of flow analysis, inductive assertion synthesis, and a language due to Dijkstra, pages 380-393. In *Program Flow Analysis: Theory and Applications*, Prentice-Hall, 1981.
10. Neil D. Jones, Automatic program specialization: A re-examination from basic principles, in D. Bjørner, A.P. Ershov, and N.D. Jones (eds.), *Partial Evaluation and Mixed Computation*, pp. 225–282, Amsterdam: North-Holland, 1988.
11. N. D. Jones,*The Essence of Program Transformation by Partial Evaluation and Driving*, in *Logic, Language and Computation, a Festschrift in honor of Satoru Takasu,* edited by Masahiko Sato N. D. Jones, Masami Hagiya, pages 206–224, S-V, April 1994.
12. Donald E. Knuth, James H. Morris, and Vaughan R. Pratt, Fast pattern matching in strings, *SIAM Journal of Computation*, 6(2), pp. 323–350, 1977.
13. Shao, Z., J.H. Reppy, A.W.Appel, Unrolling lists. In *ACM Conference on Lisp and Functional Programming*, pp. 185–195, ACM Press, 1994.
14. V.F. Turchin, 'The concept of a supercompiler,' *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, July 1986.

This article was processed using the LaTeX macro package with LLNCS style