On Static Properties of Specialized Programs*

Karoline Malmkjær †
Department of Computing and Information Sciences
Kansas State University ‡

Abstract

Specializing programs by partial evaluation is well defined extensionally, but in practice no intensional properties, such as what the specialized programs will look like or how they will run, can be ensured. We propose a technique for obtaining information about the intensional properties of specialized programs based only on the partial evaluator and the source program. This is useful, *e.g.*, when the static data are not yet available, when we specialize with respect to a large number of static data and want to predict the quality of the results, or when we wish to state formal results about the family of residual programs obtained by specializing the same source program.

The approach is based on a self-applicable polyvariant specializer and uses a grammar-generating abstract interpretation. This paper reports ongoing research, and the grammar-generating interpretation is only developed and proven safe for a subset of the target language. The remaining parts are outlined, but not completed. A simple example of a grammar representing a family of specialized programs is given to illustrate the approach.

1 Introduction

A partial evaluator is a program that takes a program and part of its input data and generates a version of the program specialized to that data. When the specialized version is applied to the remainder of the input data, it gives the same output as the original program would have given on the complete data [Bjørner, Ershov & Jones 88].

Thus the input-output function computed by the specialized version is determined by the definition of a partial evaluator. But this definition tells us nothing about the intensional properties of the specialized program, that is, what it looks like as a program: structure, size, complexity, data structures, *etc.*

*Revised version of the paper published in Bigre journal, number 74, 1991

†supported by DIKU – Computer Science Department, University of Copenhagen, Denmark
‡Manhattan, Kansas 66506, USA. e-mail: karoline@cis.ksu.edu

These properties depend on which particular algorithm is used in the partial evaluator, as well as on the source program and the given (static) part of the data. Here, we are concerned about determining this kind of properties in the case where we are given a particular partial evaluation algorithm and a particular source program, but we do not know the value of the static data. A good example is the application of partial evaluation to compilation [Futamura 71]. When we specialize an interpreter with respect to a source program, the specialized program is in effect a compiled target program. Here we consider the effectiveness of this compilation in terms of how the target programs will look, independently of any specific source program.

To see how to obtain this kind of results, let us for the moment assume that a partial evaluator takes two arguments, a source program and some static data. If we then perform a non-standard interpretation of the partial evaluator, where the abstraction of the source program is $\lambda\, x.\, x$ (the actual source program is given) and the abstraction of the static data is $\lambda\, x.\, \top$ (the static data could be anything), clearly the result of this non-standard interpretation (if it terminates) will be some abstract description of the possible specialized programs.

Although the actual design of a satisfactory non-standard interpretation for this purpose may entail many complex issues, the idea in itself is quite simple. But in order to avoid analyzing the actual partial evaluator, we are going to simplify it even further, using the idea of a generating extension [Ershov 78].

When we have a source program, `p`, the generating extension of `p` is a program, $\mathtt{G}_{\mathtt{p}}$, that takes the static data as an argument and produces the specialized version of `p`, with respect to that data, as output. In other words, $\mathtt{G}_{\mathtt{p}}$ knows how to specialize `p` and nothing else. So a non-standard interpretation of $\mathtt{G}_{\mathtt{p}}$, where the abstraction of the static data is again $\lambda\, x.\, \top$, also gives an abstract description of the possible specialized versions.

In order to ensure that the generating extension produces the same specialized program as the partial evaluator that we were originally interested in, we obtain the generating extension by self-application of the partial evaluator. This has, however, no influence on the discussions in the rest of this paper, except to the extent that our results from analyzing the generating extension are also results about the partial evaluator that produced it.

We have applied this idea to a particular partial evaluator, Similix [Bondorf & Danvy 91, Bondorf 91], that is, we have designed a non-standard interpretation for a first-order subset of the language of the generating extensions. The non-standard interpretation gives an abstraction of the output of programs in that language.

We have factorized the standard semantics into a core and a standard interpretation and we describe the non-standard semantics as a different interpretation of the core, following [Jones & Nielson 91]. This enables us to establish the desired relation between the standard and the non-standard semantics simply by proving that this relation holds between the interpretations. We have proven this for key parts of the interpretations, but the proof is not finished for the entire language. We are currently working on completing the proof as well as on the extension to higher order functions.

The next section describes the language we are analyzing and the key points about the abstraction of values and functions. Then we give a safety relation that fulfills our purpose and outline the proof that this relation is preserved. We conclude with a small example and a

$$
\begin{array}{lcl}
\mathrm{P} & ::= & \mathrm{D}_1 \ldots \mathrm{D}_n \\
\mathrm{D} & ::= & (\text{define } (\mathrm{F}\ \mathrm{I}_1 \ldots \mathrm{I}_n)\ \mathrm{E}) \\
\mathrm{E} & ::= & \mathrm{C} \mid \mathrm{I} \mid (\text{if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2) \mid (\mathrm{O}\ \mathrm{E}_1 \ldots \mathrm{E}_n) \mid (\mathrm{F}\ \mathrm{E}_1 \ldots \mathrm{E}_n) \mid (\text{let } ((\mathrm{I}\ \mathrm{E}_1))\ \mathrm{E}_2) \\
\mathrm{O} & ::= & \text{cons} \mid \text{car} \mid \text{cdr} \mid \ldots \mid \text{build-def} \mid \text{build-cond} \ldots
\end{array}
$$

Figure 1: Abstract syntax of a subset of the Similix language

$\mathcal{P}$: Program $\to$ Procedure-Name $\to$ *Val** $\to$ *Val*
$\mathcal{D}$: Definition $\to$ *Template* $\to$ *Template*
$\mathcal{E}$: Expression $\to$ *Env* $\to$ *Penv* $\to$ *Store* $\to$ *Result*
$\mathcal{O}$: Prim-Op $\to$ *Val** $\to$ *Store* $\to$ *Result*

$$\mathcal{P}[\![\mathrm{D}_1 \ldots \mathrm{D}_n]\!]\,\mathrm{F}\,v^* = \mathit{let}\ \varphi = \mathbf{mk\text{-}penv}\,\lambda\,\tau.\ \mathcal{D}[\![\mathrm{D}_1]\!](\ldots\mathcal{D}[\![\mathrm{D}_n]\!]\,\tau)\ \mathit{in}\ ((\varphi\,\mathrm{F}\,v^*\sigma_{init}) \downarrow 1)$$

$$\mathcal{D}[\![(\text{define } (\mathrm{F}\ \mathrm{I}_1\ \ldots\ \mathrm{I}_n)\ \mathrm{E})]\!]\,\tau =$$
$$\mathbf{extend\text{-}template}\,\mathrm{F}\,(\lambda\,\varphi\,(v_1 \ldots v_n)\,\sigma.\ \mathcal{E}[\![\mathrm{E}]\!](\mathbf{mk\text{-}env}\,(\mathrm{I}_1 \ldots \mathrm{I}_n)\,(v_1 \ldots v_n))\,\varphi\,\sigma)\,\tau$$

$$\mathcal{E}[\![(\text{if } \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\,\rho\,\varphi\,\sigma = \mathbf{cond}(\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\,\varphi)\,(\mathcal{E}[\![\mathrm{E}_1]\!]\,\rho\,\varphi)\,(\mathcal{E}[\![\mathrm{E}_2]\!]\,\rho\,\varphi)\,\sigma$$
$$\mathcal{E}[\![(\mathrm{O}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\,\rho\,\varphi\,\sigma = \mathcal{O}[\![\mathrm{O}]\!]\,((\mathcal{E}[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma) \downarrow 1, \ldots, (\mathcal{E}[\![\mathrm{E}_n]\!]\,\rho\,\varphi\,\sigma) \downarrow 1)\,\sigma$$
$$\mathcal{E}[\![(\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\,\rho\,\varphi\,\sigma = (\varphi\,\mathrm{F})\,((\mathcal{E}[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma) \downarrow 1, \ldots, (\mathcal{E}[\![\mathrm{E}_n]\!]\,\rho\,\varphi\,\sigma) \downarrow 1)\,\sigma$$

mk-penv : (*Template* $\to$ *Template*) $\to$ *Penv*
extend-template : Procedure-Name $\to$ (*Penv* $\to$ *Proc*) $\to$ *Template* $\to$ *Template*

Figure 2: A sample of the core semantics

generous list of future work.

2 Design of our non-standard interpretation

This section describes the language we have been working with and its standard denotational semantics. We then discuss which abstractions would be appropriate for our purposes and give the non-standard interpretation operating on these abstractions, along with the relation we wish to establish between the standard and the abstract values.

As a concrete example, we have been studying the partial evaluator Similix-2. A generating extension produced by Similix-2 is written in an untyped, call-by-value, functional language (a subset of Scheme [Rees & Clinger 86]) with a notion of global structures and extra primitives for handling syntax trees. As a first approximation, we handle only a first-order subset of this, corresponding to the syntax given in figure 1. It has the expectable semantics, so figures 2 and 3 show only the parts of interest to the abstraction. We use the usual domains of *Val*, *Env*, *Proc*, *Penv*(environment of procedures), and *Result* = *Val* $\times$ *Store*, as well as a special domain *Template* for defining procedures, with the corresponding variables v, ρ, f, φ, r, and τ. Since Similix-2 supports global structures, but not a traditional store, the domain store is just

$\varphi \in \mathit{Template} = \mathit{Penv}$
$\theta \in \mathit{Penv} \to \mathit{Proc}$
$v \in \mathit{Val} = (\mathit{Nat} + \mathit{Bool} + \mathit{Str} + \mathit{List} + \mathit{Res\text{-}Expr} + \mathit{Res\text{-}pgm})_\perp$
$\mathbf{mk\text{-}penv}_{std} = \mathit{fix}$
$\mathbf{extend\text{-}template}_{std} = \lambda\, \mathrm{F}\, \theta\, \varphi.\ [\mathrm{F} \mapsto \theta\, \varphi]\varphi$

Figure 3: A sample of the standard interpretation

$\mathit{Val} \times \mathit{Val}$. Since the evaluation order for arguments in an application is un-specified in Scheme, the store is deliberately not single-threaded in this specification.

We give the factorized semantics, to simplify the description of the non-standard interpretation. Note that the only part of the $\mathcal{E}$ valuation function that is not part of the core is the treatment of the conditional. The denotations of the primitive operators, on the other hand, have to be defined mainly in the interpretations.

To determine how to abstract the interpretation, let us consider our goals

- we are interested in the *output* of the program being analyzed, that is, not in the internal operations of the program, except where this will somehow influence the output. This differs from compiler-oriented applications of abstract interpretation, such as dependency analysis or liveness analysis.

- we are mainly interested in the *structure* of the output, considered as a syntax tree. So it is not very important whether a number is odd or even, or whether it is the result of multiplying two input values. What is important, however, is for example whether a structured value is obtained by applying the primitive `build-cond`.

To represent the structure of an output that may be constructed using recursive calls, we will use the technique of abstracting values as grammars [Jones 87]. With this technique, the output of a procedure is denoted by a non-terminal and a grammar-rule for that non-terminal describing the construction of the output, possibly using the non-terminal recursively. This provides a finite representation of a possibly infinite output.

Thus the non-standard denotation of an expression which is the body of a procedure must be the right hand side of this rule. This can be obtained by symbolically evaluating the body in an environment where the procedure denotes a constant function that always returns the corresponding non-terminal (thus ensuring termination).

The relation we expect to hold between the standard denotation and the abstract denotation is that a standard denotation must be in the language generated by the abstract denotation. This implies that the denotation of the body should also contain a grammar defining the non-terminals used in the right hand side term (including the non-terminal of the procedure itself).

So the abstraction of the domain of values must be the product of a domain of potential right hand side terms and a domain of grammars: $(\alpha, \gamma) \in \mathit{Rhs} \times \mathit{Grammar} = \mathit{Val}_{Abs}$.

To ensure that the safety relation always holds, the symbolic evaluation of the body of a procedure must take place in an environment where the procedure name is bound to a constant function returning the corresponding non-terminal *and* a grammar with a rule for

$$
\begin{array}{lcl}
P_{Val}(v,(\alpha,\gamma)) & = & v \in L(\alpha,\gamma) \\
P_{Proc}(f,f_\gamma) & = & \forall v^* \in \mathit{Val}^*_{std}, \sigma \in \mathit{Store}_{std}, v^*_\gamma \in \mathit{Val}^*_\gamma, \sigma_\gamma \in \mathit{Store}_\gamma : \\
 & & P_{Result}(f\, v^*\sigma, f_\gamma\, v^*_\gamma \sigma_\gamma) \\
P_{Template}(\varphi,\tau) & = & \forall \varphi_\gamma \in \mathit{CProc} : Q(\varphi,\varphi_\gamma) \Rightarrow P_{Penv}(\varphi, \tau\, \varphi_\gamma) \\
\text{where } Q(\varphi,\varphi_\gamma) & = & \forall \mathrm{F} \in \text{Procedure-Name} : \varphi_\gamma\, \mathrm{F} = \bot_{\mathit{CProc}} \vee P_{Proc}(\varphi\, \mathrm{F}, \varphi_\gamma\, \mathrm{F})
\end{array}
$$

Figure 4: Relations between the standard and the abstract domains

that nonterminal whose right hand side is the result of the symbolic evaluation of the body. To solve this problem we use a fixed point over the abstract values: if $[\![$(define (F I) E)$]\!]$ is a procedure definition, and the corresponding non-terminal is **f**, then the denotation of the body is

$$\mathit{fix}\ \lambda\ (\alpha,\gamma).\ \mathcal{E}[\![\mathrm{E}]\!]([\mathrm{I} \mapsto \top, \mathrm{F} \mapsto (\mathbf{f}, \mathit{add\text{-}rule}\ (\mathbf{f} \to \alpha)\ \ \gamma)]\rho\, \varphi\, \sigma$$

So our technique for terminating the non-standard interpretation is to replace recursive procedures by constant procedures returning recursively defined values[1]. This is why the application of procedures is not changed in the non-standard interpretation.

Abstracting the domain *Template* as $\mathit{CEnv} \to \mathit{Penv}$, where *CEnv* is the domain of function environments containing only constant functions, we can generalize this to several mutually recursive procedures and to include the store. This gives the following interpretations of the combinators **extend-template** and **mk-penv**:

$\mathbf{mk\text{-}penv}_\gamma = \lambda\ \tau.\ \mathit{fix}\ \tau\, (\lambda\ \mathrm{F}.\ \bot_{\mathit{CProc}})$

$\mathbf{extend\text{-}template}_\gamma =$

$\lambda\, \mathrm{F}\, \theta\, \tau.$

$\lambda\ \varphi_C.\ [\mathrm{F} \mapsto ((\varphi_C\, \mathrm{F}) = \bot_{\mathit{CProc}}) \to$

$\lambda\ v^*\sigma.\ \mathit{let}\ \alpha' = \mathit{mk\text{-}non\text{-}terminal}\ \mathrm{F}\ \sigma\ \mathit{in}$

$\mathit{let}\ r_\gamma = \mathit{fix}\, \lambda\ r_\gamma.\ \theta(\tau([\mathrm{F} \mapsto \lambda\ v^*\sigma.\ (\mathit{abstract\text{-}result}\, \alpha'\, r_\gamma)]\varphi_C))\, v^*\sigma$

$\mathit{in}\ (\mathit{abstract\text{-}result}\, \alpha'\, r_\gamma)$

$[\![\, \varphi_C\, \mathrm{F}]\, (\tau\, \varphi_C)$

The **cond** combinator is abstracted in the usual way, merging the two branches, independently of the test.

The safety relation between the standard and abstract domains of values is, as mentioned, that the standard value v is in the language generated by the abstract value (α,γ).

We denote the language generated by a grammar γ by $L(\gamma)$. By an abuse of notation, we will use $L(\alpha,\gamma)$ to denote the language generated by a term α, where the non-terminals in α may be defined in γ and if they are not, they are taken to mean $\bot$, except for the predefined non-terminal **C**, which is taken to denote the set of constants, and other similar predefined non-terminals that we use for convenience.

The relations between compound domains are built up from the relations between their components in the usual way, except P_{Proc} and $P_{Template}$, which are given in figure 4.

We have proven that **extend-template** and **cond** preserve the relation. The proof for

[1]Clearly this will only terminate if our computation of the recursive value terminates, but that is easier to ensure.

cond is quite straightforward. The proof for **extend-template** involves rewriting the standard interpretation to an equivalent interpretation using two fixed points. It then proceeds by two nested fixed point inductions. The proofs are omitted here.

While the method outlined above does give a grammar representation of the output of generating extensions, *i.e.*, of specialized programs, we are not quite satisfied with the results. The grammar is structured in a way that reflects the call-structure of the generating extension. While this is necessary, for reasons of termination, it is not sufficient. We would like the grammar to also represent the call-structure of the specialized programs.

Obtaining this is actually relatively simple in Similix-2, since the same primitive is used for constructing procedure calls and procedure definitions. The reasons for this are inherent in the partial evaluation algorithm used in Similix-2, but need not concern us now. By modifying this primitive to produce a grammar rule instead of a definition and the corresponding non-terminal instead of the procedure name in a call, the structure of the grammar will also reflect the call-structure of the generated programs. To prove that this interpretation preserves the safety relation requires a modification in the language that a grammar is supposed to generate and we have not yet completed this proof.

3 Example

Let us consider a very simple example, to show the basic features of the proposed abstraction. The source program given to the left in figure 5 takes three lists and appends the third to the end of the first and to the end of the second and then conses the two results together.

```
(define (main x y z)
   (cons (append x z) (append y z)))

(define (append l1 l2)
   (if (null? l1)
       l2
       (cons (car l1)
             (append (cdr l1) l2))))
```

```
(define (main-0 z_0)
   (cons (cons 'a (cons 'b z_0))
         (cons 'c (cons 'd z_0)))))
```

Figure 5: The source program main (left) and the version specialized with respect to (a b) and (c d) using Similix (right)

If we specialize this program for example with respect to (a b) and (c d) as its two first arguments, we get the result shown to the right in figure 5. The two calls to append have been unfolded during specialization and the static lists have been broken down into their components.

We would expect any specialization of main with respect to two static first arguments to look very similar to this: one procedure, that takes one argument and conses together two things, the first of which is the result of consing some constant values onto the argument and

```
specialize-0      ::= (define (ID-main ID-z)
                              (cons process-expr-2 process-expr-2))
process-expr-2    ::= ID-z | (cons C process-expr-2)
```

Figure 6: BNF describing the possible results of specializing append with a static first argument and a dynamic second argument

the second of which is the result of consing some other constant values onto the argument. In other words, we would expect it to be in the language generated by the BNF in figure 6.

This BNF has been obtained by hand from the generating extension for the program in figure 5 using the principles of the algorithm outlined here.

4 Related work

The present work depends strongly on previous work in abstract interpretation, mainly other work on grammar-generating interpretations [Jones 87, Mogensen 88] and the ideas of a factorized semantics presented in [Jones & Nielson 91].

To the best of our knowledge, there has been no previous work on determining intensional properties of specialized programs in a generic way.

Some partial evaluators have been proven to produce residual program with some determinable intensional properties, *e.g.*, in Similix, computations in the source program are guaranteed not to be duplicated in the residual program [Bondorf & Danvy 91]. This differs from the present approach by being a fixed property of the partial evaluator, that can be determined once and for all.

5 Conclusion and future work

We have outlined a method for generating abstract representations of specialized programs from the partial evaluator and the source program. The method is based on a grammar generating non-standard interpretation for the generating extension of the source program. The non-standard interpretation ensures termination by replacing recursive functions by constant functions returning recursive values. The safety relation has been proven for this replacement.

We are currently working on completing the proof of safety for the first order subset and on the extension to higher order functions. Afterwards we intend to implement a prototype and to refine the analysis to distinguish arguments to functions. We also plan a refinement that would produce attribute grammars, which would give a better handle on complexity properties of the specialized programs.

Acknowledgements

To David Schmidt for his encouragement and sound advice, to Olivier Danvy and to the TOPPS group at DIKU for discussions on grammars and other related topics, and to Carolyn Talcott for hosting me at Stanford while I was writing this paper.

References

[Bjørner, Ershov & Jones 88] Bjørner, D., A. P. Ershov & N. D. Jones (eds.): *Partial Evaluation and Mixed Computation*, North-Holland (1988)

[Bondorf & Danvy 91] Bondorf, A. & O. Danvy: "Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types", to appear in *Science of Computer Programming* (1991)

[Bondorf 91] Bondorf, A.: "Automatic autoprojection of higher order recursive equations", to appear in *Science of Computer Programming* (1991)

[Ershov 78] Ershov, A. P.: "On the Essence of Compilation", in *Formal Description of Programming Concepts*, E.J. Neuhold (ed.), pp 391-420, North-Holland (1978)

[Futamura 71] Futamura, Y.: "Partial Evaluation of Computation Process – An Approach to a Compiler-Compiler", *Systems, Computers, Controls*, vol 2, no 5, pp 45-50 (1971)

[Jones 87] Jones, N. D.: "Flow analysis of lazy higher-order functional languages", in *Abstract Interpretation of Declarative Languages*, Samson Abramsky & Chris Hankin (eds), pp 103-122, Ellis Horwood, Chichester (1987)

[Jones & Nielson 91] Jones, N. D. & F. Nielson: "Abstract Interpretation: a Semantics-Based Tool for Program Analysis", invited paper (in preparation) for *The Handbook of Logic in Computer Science*, North-Holland (1991)

[Mogensen 88] Mogensen, T. Æ.: "Partially Static Structures in a Self-Applicable Partial Evaluator", in [Bjørner, Ershov & Jones 88] (1988)

[Rees & Clinger 86] Rees, J. & W. Clinger (eds.): "Revised^3 Report on the Algorithmic Language Scheme", *Sigplan Notices*, Vol. 21, No 12 pp 37-79 (December 1986)