ABSTRACT INTERPRETATION OF PARTIAL-EVALUATION ALGORITHMS

by

KAROLINE MALMKJÆR

cand. scient., Københavns Universitet, 1989

A DISSERTATION

submitted in partial fulfillment of the

requirements for the degree

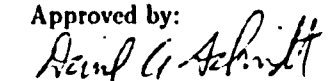
DOCTOR OF PHILOSOPHY

Department of Computing and Information Science
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1993

Approved by:



David A. Schmidt
Major Professor

Contents

List of Figures

Acknowledgements

I would like to thank the American-Scandinavian Foundation and Henrik Kauffmann's Fund for supporting my first year at Kansas State University and convincing me that it would be possible to fund my PhD studies. For support of my first year of study, I also owe thanks to the Fulbright Commission and The Danish Research Academy.

Also thanks to the IIE and in particular Mrs. Phyllis Cotten, who were helpful and accommodating in the administration of my visa, in spite of my sometimes odd travel schedule.

Although these sources were important in starting my PhD studies, the biggest thanks go to DIKU, the Computer Science Department at the University of Copenhagen, that supported the remainder of my studies, and to Neil Jones, who was my local advisor at DIKU.

Also thanks to the TOPPS group at DIKU, in particular Torben Æ. Mogensen, and also Anders Bondorf, Jesper Jørgensen, Carsten Gomard, Carsten Kehler Holst, and everybody else.

Thanks to Carolyn Talcott, for inviting me to Stanford University during the summer of 1991, and for her sound advice and her continued interest in my work.

At Kansas State University, I would like to thank George Strecker for teaching me more about cats and Julie Strecker for bringing an extra level of culture to Manhattan, Kansas. Thanks to Maria Zamfir for her kind interest and for teaching me algebraic semantics. Thanks to Austin and Sonja Melton for renting their house to me when I first arrived in Kansas, and for their continued support and friendship. Thanks to Rod Howell for his comments on my proposal and dissertation and for good lunch-time company. Thanks to Nancy Kinnersley for agreeing to be the Kansas University representative on my committee and to David Surowski for stepping in as a proxy for George Strecker at the last minute. Thanks to Beth Unger and Sandy Randel for helping me meet the administrative challenges.

Thanks to the members of the programming languages group, including the visiting mathematicians, to the lunch crowd, and to the regular, and the not-so-regular, participants to the social hour.

Many ideas in this dissertation were developed over a cup of hot tea at the Espresso Royale Caffe.

Finally, thanks to David Schmidt for welcoming me to K-State, for teaching me about the foundations of programming languages and formal semantics, and for his well-reasoned feedback to my sometimes hazy ideas.

Thanks to my parents, for bearing with my four year absence.

Thanks to Olivier Danvy, for his support and encouragement.

Chapter 1

Introduction

A partial evaluator is a program transformer that takes as input a program (the source program) and part of the input data of that program (the static data) and produces a specialized version of the source program (the residual program). When the residual program is run on the remainder of the input (the dynamic data), the result must be the same as if the source program had been run on the entire input.

Apart from the theoretical interest of such a transformation, partial evaluation is a sound technique for deriving formally correct programs. If the partial evaluator is correct, which can be proven once and for all, and if the source program is correct or definitional, the residual program is formally correct, since the residual program applied to the dynamic data must give the same result as the source program applied to all the data. This correctness follows from the definition of partial evaluator and is purely extensional.

Such a correctness is a strong argument for partial evaluation as a general program-development tool. However, to realistically apply partial evaluation to areas such as compiler generation, it is not sufficient that the generated compilers and compiled programs are correct. They are also expected to fulfill a number of intensional criteria.

Here we address the problem of determining intensional properties for any class of residual programs defined by a partial evaluator and a source program.

1.1 Outline

Partial evaluation (also known as program specialization) has been proposed as a generic technique for removing interpretive overhead [EH80]. The theoretical observation that a self-applicable partial evaluator could function as a compiler generator [Fut71], motivated the development of the first practical self-applicable partial evaluator Mix [JSS85, JSS89]. The experience from the Mix project was used to develop self-applicable partial evaluators for a first-order subset of Scheme with global variables [BD91] and for higher-order subsets of Scheme [Bon91a, Con90].

The availability of a running partial evaluator for a realistic (though limited) language prompted a flurry of application experiments. Many of these experiments are on generating compilers for different programming languages or compiling patterns and generating pattern compilers for pattern matching, but the applications also touch a number of other areas. An important issue is often to establish whether the results obtained automatically will be "as good as" some previously known structures and algorithms obtained by hand. This includes whether the generated compilers have a "natural" structure and whether compiled patterns are as simple as known pattern matching algorithms.

Generally, satisfactory results are only obtained in these experiments by "tuning" the source programs to the particular partial evaluator being used. These tunings are based on observing problems in the results and repairing them by reverse engineering. After repeated experiments, a source program is obtained that gives satisfactory results when specialized with respect to any static data.

Formally proving such a general claim about the quality of the results is messy and tedious, however. It involves the behaviour of the partial evaluator, which is theoretically simple, but as a program quite large, involving quite a lot of higher-order programming and relying on several global variables for bookkeeping and to generate unique names.

Still, it is clear that more formal statements about the results of these experiments is desirable. In a realistic application we would want a guarantee that the results of partial evaluation are in a given format, so that we can base any subsequent processing on this.

So we look for an analysis that must be correct and that gives intensional information about residual programs. That is, given a partial evaluator, a source program and a binding time pattern, describe the familily of residual programs.

We obtain this by designing an analysis that describes the output of the program being analyzed as a grammar, and applying this analysis to the generating extension of the source program.

The generating extension of a source program p is a program G_p, so that when G_p is run on the static data, it gives the specialization of p with respect to that static data. When we have a particular partial evaluator, we can obtain a generating extension that gives the same residual programs as that partial evaluator by self-application.

So by applying our analysis to the generating extension with the totally undescribed static data, we obtain a grammar that describes the family of residual programs.

We design our analysis as an abstract interpretation based on denotational semantics. We use a depth-first strategy in analyzing procedure calls, analyzing the callee before completing the analysis of the caller. This corresponds to the development in partial evaluation, where the breadth-first strategy of Mix was replaced by a depth-first strategy in Similix and Schism. Instead of the usual global fixed-point of abstract interpretation, we use local fixed-points for each procedure. This facilitates a proof by logical relations [JM86b, Nie89].

1.2 Overview

The next chapter describes the framework that we are working in: partial evaluation of applicative, call-by-value languages and abstract interpretation based on denotational semantics. The particular partial evaluator, Similix, that we will use as the main example, is also described.

Chapter 3 gives examples of applications of partial evaluation and describes the kind of properties we would like to determine. It discusses in a general way what form the properties might be described in and how they could be determined.

Chapter 4 gives actual analyses for predicting intensional properties of residual programs, both for first-order and higher-order Similix.

Several examples of the result of analysis are presented in chapter 5. The first-order analysis is tested on the so-called "MP-interpreter" from the Similix distribution package and on a pattern matching example. To show that the analyses have a more general application, we also give an example unrelated to partial evaluation: the CPS-converter of Danvy and Filinski [DF92]. The CPS-converter is an ideal example, since it is very compact but it still produces syntax as output. Furthermore it relies heavily on higher-order programming, demonstrating this aspect of the algorithm.

In chapter 6 both algorithms are proven safe with respect to a standard semantics, using logical relations.

Chapter 7 describes the implementation of the first-order algorithm. The implementation is a prototype, designed to test the quality of the output from the analysis rather than for efficiency. The implementation follows the formal algorithm very closely.

Chapter 8 discusses several possible modifications of the analyses, either for the purpose of speeding up analysis or for the purpose of improving precision. Not surprisingly, these modifications tend to counteract each other.

In chapter 9 we discuss related works, both in partial evaluation and in abstract interpretation.

Chapter 10 is the conclusion.

Chapter 2

Background: Partial Evaluation and Abstract Interpretation

2.1 Partial Evaluation

In this section we give a brief outline of the field of partial evaluation, its theoretical foundation in Kleene's S_n^m theorem, the application to compiler generation based on self-application (the Futamura projections) and the practical algorithms that have been developed. We will focus on algorithms and techniques for applicative languages. Many of these carry over to imperative languages, while logic programs require somewhat different techniques.

2.1.1 Introduction

Partial evaluation is the process of automatically specializing programs. That is, given a program and the values of some of its input, to produce another program, the specialized (or *residual*) program, with the following property: running the specialized program on a set of values for the remaining input gives the same result as running the original program on all the values. A program that does this is called a ***partial evaluator*** or a ***program specializer***. If we have a partial evaluator MIX written in L, the definitional property can be expressed in the following way[1]:

$$\forall p, s, d : (\!|p|\!)\,[s, d] = (\!|(\!|MIX|\!)_L\,[p, s]|\!)\,d$$

The name program specialization has also been proposed instead of partial evaluation, since it describes the purpose, rather than the method, but has not been able to replace the older term. The concept of specialization is well-known from mathematics. When given a function of two variables $f(x, y)$, we readily refer to f_{x_0} as the function of one variable that for any y_0 gives the result $f(x_0, y_0)$.

Specializing a program with respect to part of its input is interesting because the specialized program may run faster than the original program. This is an advantage if we want to run a program many times on input that is partly identical or if some of the input is known in advance and the process is time-critical when the last input arrives. Examples of this reach from interpreters, that we want to run many times on the same source program, to the xphoon program in the X Window System[2], where the bitmap representing the picture of the moon is "compiled in"[3] to the program

[1] We use $(\!|p|\!)_L$ for the function computed by program p, when considered to be a program in language L. $(\!|p|\!)_L\,v$ is the result of applying that function to argument v, or, in other words, the result of running p on v.

[2] X Window System is a trademark of the Massachusetts Institute of Technology.

[3] to quote the xphoon manual page

loading the bitmap.

Terminology and notation

We usually refer to the input program of the partial evaluator as the *source program*, and to the output as the *residual program*.

In languages where there is no obvious main procedure, we also need to specify which of a program's procedures should be specialized. This procedure is usually called the *goal procedure*.

When a program is being first partially evaluated and then executed, it is, in a sense, being executed in several "stages". To distinguish these stages, the notion of *binding-time* has been imported from compiler technology and extended.

The binding-time of a variable is the time at which a value can be assigned to it. In compilation, one might have, for example, compile-time, link-time, run-time.

In partial evaluation, we consider two binding-times for each program. We say that a variable s is *static* if it can be assigned a value at partial evaluation time and that it is *dynamic* if it might not be possible to assign a value before running the residual program. If we consider the parameters of the goal procedure, for example, the parameters corresponding to known arguments, that are given to the partial evaluator, are static, and the parameters corresponding to unknown arguments are dynamic.

The concept of binding-times is extended from variables to all the syntactic categories of the programming language in the obvious way: if an expression can be computed at partial evaluation time, it is said to be static. If it might depend on some dynamic variables and can only be computed at run-time, it is said to be dynamic.

In order to reduce the inevitable confusion of levels when discussing programs as data objects and the semantics of program transformers, we will introduce some restrictions on standard terminology. We will use the word *function* exclusively to refer to semantic and mathematical functions. We will refer to unnamed lambda abstractions in programs as *lambda abstractions* and to named and unnamed lambda abstractions (including those defined with the syntax `(define (foo args) body)`) as *procedures*.

As a notational convenience, we will use `typeface` for program text. This indicates that the program should be understood as syntax[4]. All semantic and mathematical formulae, as well as meta-variables, will be in *italics*. Thus we might use the meta-variable P to denote the program `(define (foo a) (add1 a))`. When we want to refer to the function computed by program P, we will enclose it in brackets: $\langle\!|P|\!\rangle$ or $\langle\!|$`(define (foo a) (add1 a))`$|\!\rangle$. In this case that would be the function $\lambda x.(1 + x)$.

To distinguish between the how and the what of programming, we adopt the terminology of Talcott [Tal85] and use the word *intension* for those aspects of a program that concerns its text and execution, while the input-output function that a program computes is considered an *extensional* aspect.

2.1.2 The formal basis: Kleene's S_n^m-theorem

Kleene's S_n^m-theorem is considered to be the theoretical basis of partial evaluation, since the theorem states that a partial evaluator for the λ-calculus exists and is computable:

Theorem 1 *[Kle52]*

[4] We make an exception for the lambda-calculus, though, where we always give terms in mathematical font.

For all m, n there exists a computable function, S_n^m, of $m+1$ arguments, so that for all terms $f, x_1, \ldots, x_{m+n}$:

$$(\!|f|\!)_\lambda[x_1, \ldots, x_{m+n}] = (\!|S_n^m[f, x_1, \ldots, x_m]|\!)_\lambda[x_{m+1}, \ldots, x_{m+n}]$$

The proof basically amounts to currying f and constructing the application of f to its first m arguments. Computability follows, since the proof specifies an algorithm.

Example

If we have the λ-term $\lambda\ [f, g, h].\ fh(gh)$ then we can apply the function S_2^1 to this term and for example the term $\lambda\ x.\ x$, giving

$$S_2^1[\lambda\ [f, g, h].\ fh(gh), \lambda\ x.\ x] = ((\lambda\ f.\ \lambda\ [g, h].\ fh(gh))(\lambda\ x.\ x))$$

2.1.3 Algorithms for partial evaluation

For any practical application in computer science, a more efficient algorithm than the one given in the S_n^m theorem will be needed. This section outlines some algorithms developed in the area of partial evaluation.

When we are given a program and the value of its static arguments we can clearly propagate the values of the static arguments through the program and resolve all tests that rely only on the static arguments. This is also known as *constant propagation*. All non-trivial partial evaluators use this technique.

Example

If we have the Scheme program

```
(define (foo x y)
    (+ (* x x) (* y y)))
```

and specialize it with respect to x being 3, we can get[5]

```
(define (foo-0 y_0)
    (+ 9 (* y_0 y_0)))
```

The technique falls short, however, as soon as we meet a loop (iterative or recursive) that modifies the static arguments.

This can be solved by *unfolding* the loop.

Example

If we have the Scheme program

```
(define (main a b c)
    (cons (append1 a c) (append1 b c)))

(define (append1 l1 l2)
  (if (null? l1)
      l2
      (cons (car l1) (append1 (cdr l1) l2))))
```

[5]Note that all variables are renamed to avoid potential name-clashes.

and specialize it with respect to a being the list (a b) and b being the list (c d), we get

```
(define (main-0 c_0)
    (cons (cons 'a (cons 'b c_0))
          (cons 'c (cons 'd c_0)))))
```

Here the calls from main to append1, as well as the recursive calls in append1, have all been unfolded, leaving a relatively compact specialized program.

However if the loop is not controlled by static variables only, the unfolding is unbounded and the partial evaluator will go into an infinite loop. Detecting this is undecidable in general, but practical approximations (safe or – more commonly – unsafe) can be made [Jon88].

The λ-mix algorithm

[Gom89] presents a partial evaluator for a small higher-order language (essentially the simply typed λ-calculus) that mainly relies on the techniques outlined above. Furthermore, the specialization algorithm relies on previous binding-time annotations. The annotations direct which expressions can be simplified based on the static data and which expressions involve dynamic data and must be reproduced in the output. The partial evaluator uses constant propagation and unfolds applications of simple and recursive functions if they are so annotated. It also uses renaming to avoid capturing variables. The partial evaluator can be compared to an interpreter for a two-level language such as used by Nielson and Nielson [NN86].

Polyvariant specialization

One of the most popular algorithms for program specialization is known as *polyvariant specialization* [Bul84]. It is based on the notion of *program point*. Except for the obvious constant propagation and loop unfolding that can be performed, a polyvariant specializer identifies a set of program points in the program being specialized, related to the control structure of the program. Each program point is then specialized into several different versions, one for each set of static values that can be reached at that point. How the program points are selected depends on the source programming language. In simple imperative languages, labels are usually chosen as program points. In applicative languages, procedure definitions are the most common choice.

The polyvariant specialization algorithm works by keeping a list of program-point/argument pairs and the corresponding residual (specialized) program points. When a call is encountered, it is compared to the entries in the list. If it is found, a residual call to the residual program point is generated. If it is not found, a unique name is generated for the corresponding residual program point and a new entry is made in the list.

At this point, the partial evaluator has to do two things: generate a residual call and generate the text of the new residual program point. There are of course two ways of going about this, depth-first or breadth-first[6]. The original Mix [JSS89] used the breadth-first algorithm. It adds the program-point/arguments to a list (the so-called pending list) and then generates the residual call. The general algorithm loops over the pending list until it is empty. Newer partial evaluators, such as Similix and Schism, use the depth-first algorithm, first generating the residual program point and then, when returning, generating the actual residual call to the new program point.

[6]On a parallel architecture, there is also the third possibility: create two tasks and send them to two processors. This was proposed and implemented by Consel and Danvy in 1990 [CD90b]

Example

If we have the Scheme program from the previous example and specialize it with respect to c being the list (x y), then clearly we do not want to unfold the procedure calls. However, if we say that main and append1 are program points, then we can produce specialized versions of these program points with respect to the list (x y).

```
(define (main-0 a_0 b_1)
    (cons (append1-0-1 a_0) (append1-0-1 b_1)))
(define (append1-0-1 l1_0)
    (if (null? l1_0)
        '(x y)
        (cons (car l1_0)
              (append1-0-1 (cdr l1_0))))))
```

Note that an algorithm based on this principle will loop on some input because it tries to create infinitely many specialized program points. It would not be a solution to unfold the program point instead, as this would just lead to infinite unfolding. Forcing a change in the binding-times of some of the variables, however, might cause the specialization to terminate. Depending on circumstances, it is possible either to change static variables to be dynamic (note that specializing a totally dynamic program always terminates) or to change dynamic variables to be static, as shown by Barzdins [Bar91]. But determining whether a particular division of variables into static and dynamic causes the specializer to terminate is undecidable [Jon88].

Driving and the Refal system

One of the pioneer systems in partial evaluation is supercompilation [Tur79, Tur86b, Tur86a, Tur89], which uses somewhat different principles than the polyvariant specializers. It is based on the language Refal, which is usually considered a functional language, but with so strong pattern matching facilities that a comparison with some aspects of logic programming seems in order. The specialization technique employed is called "driving" and can roughly be described as a complete unfolding of the call-graph, with comparisons at each call to determine whether a "similar looking" call has already been encountered, in which case the processing of the two calls is "generalized" [Tur88] to avoid looping.

2.1.4 Self-application and partial evaluation for compiler generation

If a partial evaluator is written in the same language as it takes as input, it can be applied to a representation of itself.[7]

This can be used for compilation of programs and even for generating compilers, given an interpreter for the language we want to compile. The equations stating these results were first presented by Futamura [Fut71] and are usually known as the Futamura projections.

Before we present these results, which are essential to the current popularity of partial evaluation, let us first recall that these, like almost all theoretical partial evaluation results, are purely extensional. They simply state that given a program in some source language S, we can obtain a program in some target language T, computing the same function. This is quite independent of whether the source language is more complex than the target language or vice versa, and we have no guarantee that running the target program will be faster than interpreting the source program.

[7] This is because partial evaluation is a syntactic transformation, so the partial evaluator considers its input to be first-order: a piece of syntax, independently of which function the input computes when run.

Compilation by partial evaluation is based on the observation that an interpreter for a language S is a program *int*, written in some language L, taking two arguments, an S program and its input. The interpreter fulfills the property that

$$(\!|int|\!)_L\ [p, d] = (\!|p|\!)_S\ d$$

that is, the function computed by the interpreter, applied to the program and the data gives the same result as the function computed by the program applied to the data.

A compiler from S to T, on the other hand, is a program *comp*, written in some language L, that takes one argument, an S program, and produces a T program as output, so that the following property is fulfilled

$$(\!|(\!|comp|\!)_L\ p|\!)_T\ d = (\!|p|\!)_S\ d$$

that is, the function computed by the compiled program in T is the same as the function computed by the source program in S.

Let us assume that we have a partial evaluator *MIX*, written in a language L, specializing programs written in L, and producing residual programs written in T. By definition $(\!|MIX|\!)_L$ takes two arguments, a program p in L and some partial input of p and returns a result fulfilling:

$$(\!|(\!|MIX|\!)_L\ [p,s]|\!)_T\ d = (\!|p|\!)_L\ [s, d]$$

If we specialize *int* with respect to an S program p, the result is a T program t, so that when we run t on input d, we get the same result as if we had run *int* on $[p, d]$. In other words t fulfills the following property:

$$(\!|t|\!)_T\ d = (\!|(\!|MIX|\!)_L\ [int,p]|\!)_T\ d = (\!|int|\!)_L\ [p, d] = (\!|p|\!)_L\ d \tag{2.1}$$

So we have in effect compiled p from S into T.

Furthermore, since *MIX* is also written in L, we can specialize *MIX* with respect to *int*, giving a program c so that:

$$(\!|c|\!)_T p = (\!|(\!|MIX|\!)_L\ [MIX,int]|\!)_T p = (\!|MIX|\!)_L\ [int, p] = t \tag{2.2}$$

that is, c is a compiler from S to T.

If *MIX* is proven correct with respect to the definition of a partial evaluator and *int* is a definitional interpreter, then c is a correct compiler. This is the reason partial evaluation can be seen as a compiler generation tool.

The equations 2.1 and 2.2 are the first and second Futamura projections.

We can also forget about interpreters and compilers, however, and describe the equations only in terms of specialization. Then we observe that c is a program with the property that when it is applied to the static data of *int*, it produces the specialization of *int* with respect to that data. A program with this property (whether it is hand-written or obtained automatically) is named a *generating extension* of *int* by Ershov ([Ers78]). In general we note that whenever we have a program p, written in L, we can obtain a generating extension for p by specializing the partial evaluator with respect to p.

Definition 1 *When we have a source program, p, the* generating extension *of p is a program, G_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, G_p knows how to specialize p and no other program.

Note that although we generally talk about "the" generating extension, there are in fact many programs fulfilling this definition and they do not even produce the same residual programs. The generating extension obtained by specializing the partial evaluator with respect to p, that is, by self-application, however, produces the same specialized programs as the partial evaluator. This is natural, since it is a specialized instance of the partial evaluator, and it can easily be deduced from the definition.

2.1.5 Binding-time analysis for efficient self-application

Although self-application is technically possible whenever a partial evaluator is written in its own input language, the first results of self-applying a partial evaluator were huge and complex.

This is now recognized as being related to the fact that the partial evaluator is designed to take a program and *any* possible subset of its data. In this way it corresponds to several S_n^m functions. So for example in the case of c, computed by self-application above, c is not only a compiler taking S programs and producing T programs expecting data. It could also take the data and produce a T program expecting an S program to produce the result – a kind of "data compilation". Or it could take *both* the S program and its input data and produce a T program that, when run on no input, would produce the result. Finally, given no input, c would produce a T program expecting both an S program and its data, that is, an S interpreter written in T.

This is a purely extensional reason why the first results were too big: they simply computed a function that was too general. The solution to this problem is to say which arguments will be static at self-application time. So to produce a compiler, the partial evaluator is specialized with respect to the interpreter and the information that the first argument of the interpreter will be static and the second will be dynamic.

However, because of the limitations of the partial evaluation algorithms in use today, partial evaluators that have been successfully self-applied go one step further; this time for intensional reasons.

Any partial evaluator must have a strategy to determine whether a syntactic form depends only on static data and can be reduced or whether it may depend on dynamic data and has to be rebuilt in the specialized program.

As an example of an extremely conservative strategy, the algorithm used in the proof of the S_n^m theorem safely approximates that all reduction may depend on dynamic data and so the entire source program appears in the result.

Clearly it is possible to do better, but then the partial evaluator becomes slower. This is comparable to the usual tradeoff: fast compilation/slow runtime *vs.* slow compilation/fast runtime. In a practical partial evaluation the decision reduce/rebuild must be taken for each separate syntactic form in the source program every time it is encountered.

In self-application, the code for performing this decision is specialized with respect to each node in the syntax tree of the source program. The decision cannot be made at self-application time both because it might vary depending on the actual values and because of limitations in the information propagation in the algorithm. So each specialized version will be a conditional and each branch of this conditional will again contain the specialized version for each subnode, that again ...

This leads to unacceptably large and redundant specialized programs.

This has been solved by approximating this decision in a pre-analysis known as *binding-time analysis*. The binding-time analysis annotates each node as static (definitely and always static) or dynamic (possibly dynamic). The partial evaluator simply checks the annotation. At self-application

time, both the partial evaluator and the program are annotated, so the decision can be completely reduced and does not appear in the generating extension.

The pre-analysis also determines whether a program point should be specialized or unfolded. A procedure can safely be unfolded by the specializer for example if it does not contain any recursive calls. In case of doubt, the program point is specialized. This leads to corridor calls in the residual program, that is, procedures that are only called once. These are usually unfolded in a post-processing phase.

Most of the development in this section was inspired by [Mog89]. Reduce/rebuild is introduced as a fundamental concept in [CD90a].

Binding-time analysis of higher-order programs

Binding-time analysis for a first-order functional language can be formulated as a data-flow analysis over a two point domain (static, dynamic).

For a higher-order language, things get considerably more complicated, and this seems to be the main hurdle to overcome in extending practical partial evaluation from first-order to higher-order.

In 1989, Similix was extended to higher-order programs, using a binding-time analysis based on a pre-phase closure analysis ([Bon91a]). Schism was also extended to higher-order [Con90] using a kind of closure analysis. The closure analysis of Schism was integrated in the binding-time analysis along with a treatment of partially static structures. (See also section 2.2.5.)

A completely different approach was taken by Gomard ([Gom90]) in the lambda-mix system, where he used a binding-time analysis based on type inference. With this approach, static and dynamic are seen as two-level types, and a modified version of algorithm W is used to type the program. This approach was extended in [Hen91] and has received much attention lately, as it seems to allow considerably faster algorithms.

Either way, binding-time analysis is essentially a branch of abstract interpretation, and we will cover the topic in more detail in the next section.

Relating binding-time annotations to two-level languages

The binding-time annotated languages that the self-applicable partial evaluators take as source languages strongly resemble the two-level languages of Nielson and Nielson [NN86]. The view of a partial evaluator as a compiler reinforces this comparison, since the static expressions are "compile-time" and the dynamic expressions are "run-time". However, the two approaches differ in the treatment of the expressions as well as in their view of what the expressions are supposed to represent. A quick slogan might be that Nielson and Nielson essentially consider a two-level program to be what a partial evaluation person would call a generating extension of the original program. The static parts are converted into combinators, that can be assigned an interpretation and executed, giving, for example, a compiled program as result.

This view of a generating extension was also taken in the Holst and Danvy work on specialization by macro-expansion [HD89].

2.1.6 The notion of "specializing well"

Since a polyvariant specializer is a rather simple tool it is possible that the result of specialization looks more like the result in the proof of the S_n^m theorem than like specialized code. We say that the source program "specializes badly". The reason for such bad results is often found, not in the

Pr ∈ Program, PD ∈ Definition, F ∈ Filename, E ∈ Expression, C ∈ Constant,
V ∈ Variable, O ∈ OperatorName, P ∈ ProcedureName,
SC ∈ SimpleConstant, Num ∈ Number, Bool ∈ Boolean,
Str ∈ String, Ch ∈ Character, Sym ∈ Symbol, Q ∈ QuotedValue, Pa ∈ Pair, Ve ∈ Vector

```
Pr   ::= (loadt F)* PD+
PD   ::= (define (P V*) E)
E    ::= C | V | (if E E E) | (let ((V E)) E)
       | (begin E+) | (O E*) | (P E*) | (lambda (V*) E) | ( E E*)
C    ::= SC | (quote Q)
SC   ::= Num | Bool | Str | Ch
Q    ::= SC | Sym | Pa | Ve
Pa   ::= ( Q . Q)
Ve   ::= #(Q+)
```

Figure 2.1: Syntax of the Similix source and target language.

functionality of the source program, but rather in its structure. In one way or another, the structure of the program blocks the flow of static data.

A simple example of this is the Scheme expression `(add1 (if test 0 1))`. If we want to specialize this in a situation where `test` is a dynamic variable, we have to rebuild the conditional expression, so the argument of the operator `add1` is also dynamic. If we transform the expression to the equivalent `(if test (add1 0) (add1 1))`, however, the arguments of both the operators are now static. So even though `test` is still dynamic and we still have to rebuild the conditional, we can reduce the operations, giving the simplified `(if test_0 1 2)`.

Unfortunately we cannot just perform this transformation in all cases and obtain an optimal result. For example if one of the branches of the conditional causes an infinite loop and the test somehow avoids this, then the partial evaluator would loop if we perform the transformation, whereas the untransformed version would terminate, both at specialization time and at run time.

The notion of specializing well is addressed as a practical issue in many of the papers on partial evaluation. Consel and Danvy propose a more general technique for obtaining programs that specialize well based on continuation-passing style transformation [CD91].

2.1.7 A polyvariant specializer: Similix

Similix [BD91, Bon91a, Bon91b] is a depth-first polyvariant specializer for a subset of Scheme [CR91], using binding-time analysis for efficient self-application. In order to facilitate programming it handles global structures and a weak notion of abstract data types by which the user can extend the set of primitive operations in the source and target language. These features are used extensively in the programming of the actual self-applicable specializer and thus also appear in any generating extension.

Source and target languages

The source language of Similix is essentially a significant subset of Scheme. A BNF of the language is given in figure 2.1[8].

[8] This BNF is presented in the Similix manual [Bon91b]

The target language is the same as the source language, but certain forms are obtained by post-optimization, after the actual specialization is finished.

Phases

Partial evaluation in the Similix system consists of a large number of phases, only one of which is the actual specialization. Basically there are 5 phases: parsing, analysis, specialization, post-optimization and "un-parsing".

The parsing and un-parsing phases are just the usual translations between textual programs and the internal syntax tree representation.

The analysis phase is conceptually the most complex, consisting again of several subphases, some of which perform fixed point iteration to obtain the abstract results. The analysis phase adds several annotations to the syntax trees. The uniqueness analysis counts how often a variable is used in the body of its procedure, and if it is used more than once, a let-expression is inserted to avoid computation-duplication by the specialization. The closure analysis labels all the named procedures and lambda abstractions of the program. It then determines, for each application in the program, which procedures (either named or lambda-abstractions) might be applied there in some evaluation of the program. The binding-time analysis uses this to determine the static/dynamic annotations.

The specialization phase follows these annotations while performing the actual specialization: evaluating static expressions, rebuilding dynamic expressions, unfolding or specializing procedure calls.

The post-optimization phase performs some simple optimizations, such as removing redundant let-expressions (such as `(let ((x1 x2)) ...)`) and unfolding corridor calls (eliminating procedures that are called only once).

Global variables and abstract data types

Similix supports a weak notion of abstract data types. The user can extend the set of primitives in the source language by defining them in a so-called "adt" file. The definitions are written in Scheme and may define global structures and use side-effects on them. The side-effects cannot appear at the actual source language level, so for example a reference to a variable declared globally by a primitive will result in a complaint from Similix about an undefined variable. Any primitive defined using Scheme side-effects must furthermore be declared "opaque" to ensure proper specialization.

This feature is used in the self-applicable specializer itself. The specializer operates on programs represented as syntax trees, so the basic Scheme subset is extended with a large number of primitives for accessing and constructing syntax trees.

Furthermore, the basic administrative data-structures of Similix are coded as global structures. These include the list of procedure calls that have already been specialized and their residual names (the so-called "seen-before-list") and the already completed parts of the residual program. Primitive operations defined on the seen-before-list include the key operation **genprocname**, that takes a procedure name and a list of actual arguments and check if they are already in the list. If so, the residual name found in the list is returned along with a tag `#t`. If not, a new residual name is generated and a new entry is made in the seen-before-list. The new name is returned with the tag `#f`.

Both the seen-before-list and residual program also appear in the generating extensions produced by Similix.

Generating extensions produced by Similix

A generating extension in Similix is obtained by applying the specializer to an annotated version of the specializer, with an annotated version of the source program as static data. None of the other phases are thus included in the self-application.

This is what makes self-application manageable, but this also means that the generating extension produces residual programs that are not post-optimized. So in order to get the same residual program as one would from normal specialization, one has to run first the generating extension and then the post-optimizer and un-parser.

This also means that a generating extension is generally rather large and unwieldy, since it contains code to generate all the corridor calls and trivial let-expressions that will be unfolded in post-processing.

It is a general observation that when a program is specialized, the residual program will be structured like the static data, but use the "terminology" of the source program. This is particularly apparent when specializing an interpreter with respect to a program.

In the case of producing a generating extension by specializing the specializer with respect to a program, this effect also shows up. The generating extensions produced by Similix will use the same data-structures and the same procedure and variable names as the specializer, but will reflect the call-structure and static computations of the program.

2.2 Abstract Interpretation

2.2.1 Origin

Abstract interpretation is one of the terms used for the idea of symbolically running a program without its input data or with only an abstract description of the data. The idea seems to have originated with compiler writers (where it was named data flow analysis) and is mainly used to determine whether various optimizing transformations can safely be performed. Abstract interpretation emphasizes the formal safety of the results with respect to a standard semantics.

2.2.2 Formalization

The data flow analyses used in compilers were formalized by Cousot and Cousot [CC77]. Their formalization covers imperative languages and is based on an operational semantics for the language being analyzed. The semantics uses state transitions and is first generalized to log sets of states for each point in the program[9]. By computing the fixed point over this log, one obtains a description of all possible states that might occur during execution of the program. The generalization is then abstracted to finite abstractions of these sets, so that the analysis terminates. The correctness of the analysis is guaranteed by proving a safety condition between the set of states and their abstraction.

The safety condition is expressed in terms of the *abstraction* and *concretization* functions. The abstraction function α goes from the power set of states $\mathcal{P}(S)$, organized as a lattice by subset inclusion, to the (finite) lattice of abstract states A. The concretization function γ goes from A to $\mathcal{P}(S)$. Both α and γ must be monotone. The safety condition states that if we have a function f in the accumulating semantics and we have a function $f^{\#}$ that we want to use in the abstract interpretation instead of f, then for all sets of states $s \in S$, it must hold that $\gamma(f^{\#}(\alpha(s)) \supseteq \{f(v) \mid v \in s\}$.

[9] This generalized semantics was originally named "static", then "collecting" and sometimes "accumulating".

Abstract interpretation was extended to the general framework of denotational semantics by Nielson [Nie82].

2.2.3 Termination

An abstract interpretation is expected to always terminate. This can be obtained in several ways. The most common technique is to design the analysis to be increasing on some finite lattice or domain. A weaker version of this is simply to ensure that the domain has no infinite chains.

In cases where we cannot obtain this, Cousot and Cousot [CC77] propose to introduce an operator in the abstract interpretation (the so-called widening operator) with the property that any infinite sequence obtained by consecutive applications of the operator cannot be increasing.

2.2.4 Analysis for functional languages, minimal function graphs

Mycroft [Myc81] extended the framework of Cousot and Cousot to applicative languages by representing the abstract meaning of a procedure as a function on abstract domains. This required the design of power-domains and in cases where the ground domains were not tiny, lead to very large abstract domains.

As an alternative, Jones and Mycroft [JM86a] proposed a framework for analyzing functional languages based on logging the argument-result pairs for each procedure in a program. The minimal function graph semantics (their version of a collecting semantics) logs sets of such pairs corresponding to all possible executions by taking a fixed point over the log. For each particular analysis, an appropriate abstraction of the sets and corresponding definitions of the basic functions are chosen. In particular, the standard interpretation is a special case of the collecting semantics.

If the domain of values has no infinite chains then the log of argument/result pairs also has no infinite chains and the analysis will terminate if the basic functions are monotonic.

The partial evaluator Mix [JSS89] is given as an example of an abstract interpretation that is not guaranteed to terminate.

2.2.5 Closure analysis

In order to extend first-order analyses to higher-order languages (actually, Scheme), Shivers [Shi88, Shi91] proposed the so-called closure analysis, that for every application point gives a safe approximation of which procedures could be applied there during any execution of the program.

The analysis is a typical abstract interpretation. In pre-processing, each lambda abstraction and application point is assigned a unique label. The analysis symbolically executes the program on a domain of labels. In effect, this means that first-order values are ignored in the execution. Whenever an application is reached, the current value of of the operand is logged. By taking a fixed point over the log, an approximation of the possible procedures at each application point is obtained. Termination is guaranteed by finiteness of the domains (since there are no ground values).

Once a program is tagged with this information, an essentially first-order analysis can be used on it.

The program to be analyzed is first transformed into continuation-passing style. This simplifies the analysis, since all calls are tail-calls, so there is no need to distinguish between depth-first and breadth-first treatment of calls.

The framework allows several levels of detail in the treatment of environments, with the usual precision/complexity tradeoff.

The simplest version (called 0CFA) uses one global log that for each variable saves the lub of all values it gets bound to. This is enough to produce a general call-graph for many applications, but in higher-order procedures like `map`, it causes confusion of the sets of arguments and a serious loss of information.

A more detailed version (called 1CFA) allows to distinguish between call points, so that a variable in the environment gets bound to a call point, and a global log takes variables and call points to the lub of the values encountered at that call point.

Sestoft [Ses91] presents a similar analysis, not using continuation-passing style.

2.2.6 Generating grammars as abstract results

The need to come up with a finite abstraction of the usually infinite domains of the semantics in order to make the analysis terminate puts some severe restraints on what kind of information can be collected by an abstract interpretation. In order to stretch these restraints as much as possible, *Jones [Jon87] proposes to use grammars as finite representations of possibly infinite structures.*

The basic idea of this approach is that a non-terminal is generated for each procedure and possibly for each formal parameter in the program being analyzed. The non-terminals are intended to represent the result of procedure applications and identifier references. Correspondingly, the right hand side of a rule for a procedure non-terminal is intended to describe the result of the procedure. The right hand side of a rule for a parameter non-terminal is intended to describe the possible, corresponding, actual arguments. This way any recursive references will appear as recursive productions in the grammar.

Termination is obtained by noting that if we have a call to a procedure P in the body of a procedure Q, we can use the non-terminal of P instead of the result of evaluating P. For example the result of the non-terminating procedure `(define (P v) (cons 1 (P v)))` can be accurately (if not very usefully) described by the grammar $P \rightarrow 1 :: P$, instead of the infinite sequence $1 :: 1 :: 1 :: 1 :: 1 :: \ldots$.

Note that this approach can be compared to the minimal function graphs, only the minimal function graph analysis does not generate non-terminals for arguments of procedures.

This approach has been used for a variety of properties.

Mogensen uses a grammar representation of partially static structures in binding-time analysis. Partially static structures are data structures that are composed of both static and dynamic parts [Mog88]. In the cases where this follows a regular pattern, such as a list of pairs, where the first elements are all static and the second elements are all dynamic, it is possible to maintain that expressions involving only the first elements are static. Mogensen extends the usual binding-time domain $\{S, D\}$, $S \sqsubseteq D$, to Lisp S-expressions by adding a pairing operator P. He extends further to downwards closed (or left-closed) sets of such terms, since for partial evaluation purposes, a "worst" element is intended to represent all "better" elements as well. To get a finite approximate representation of such sets, he then introduces a grammar representation where each right-hand side can only use the pairing operator once.

An analysis that constructs such grammars from a (first-order) program is then given. It uses a non-terminal for each procedure and variable in the program and takes a fixed point over the grammar adding procedure-name/result rules and variable/value rules. The abstract cons operation uses the pairing operator, while the abstract car and cdr operations map S to S, D to D, $P(t_1, t_2)$ to respectively t_1 and t_2 and looks up a non-terminal in the grammar to destruct the corresponding right-hand side.

Termination is achieved since for each program there is only a finite set of non-terminals and thus only a finite set of possible grammars.

A grammar representation of contexts is used with a backwards analysis to determine liveness by Jensen [JM90].

A backwards analysis with a grammar representation of definition-use paths is used by Gomard and Sestoft to detect variables that can be globalized [GS91].

The binding-time analysis of Consel ([Con90]) also use this principle, both in the handling of partially static structures and in the handling of higher-order values. In this analysis, however, the non-terminals of the grammar are not just the procedures (named or lambda abstractions) of the program, but also the "cons-points" (the points in the program where a cons occurs).

2.2.7 Relational approach

In many cases, an abstract interpretation can be formulated and proven safe more simply by *factorizing* the standard semantics into a *core*, defined in terms of some unspecified domains and combinators operating on them, and an *interpretation*, defining the domains and the combinators.

Consider, for example, the following fragment of a denotational semantics, giving the denotation of arithmetic expressions over variables:

$$\begin{array}{ll} \mathtt{Expr} ::= \mathtt{I} \mid \mathtt{E}_1 \texttt{ + } \mathtt{E}_2 \mid \mathtt{E}_1 \texttt{ - } \mathtt{E}_2 & \mathcal{E} : \mathrm{Expr} \to \mathit{Env} \to \mathit{Nat} \\ v \in \mathit{Val} = \mathit{Nat} & \mathcal{E}[\![\mathtt{I}]\!]\rho = \rho\,\mathtt{I} \\ \rho \in \mathit{Env} = \mathrm{Id} \to \mathit{Val} & \mathcal{E}[\![\mathtt{E}_1 \texttt{ + } \mathtt{E}_2]\!]\rho = \mathcal{E}[\![\mathtt{E}_1]\!]\rho + \mathcal{E}[\![\mathtt{E}_2]\!]\rho \\ & \mathcal{E}[\![\mathtt{E}_1 \texttt{ - } \mathtt{E}_2]\!]\rho = \mathcal{E}[\![\mathtt{E}_1]\!]\rho - \mathcal{E}[\![\mathtt{E}_2]\!]\rho \end{array}$$

We can factorize this over the domains *Val* and *Env* with the combinators ***lookup***, ***plus***, and ***minus*** in the following way:

$$\begin{array}{lll} v \in \mathit{Val} & \mathcal{E} : \mathrm{Expr} \to \mathit{Env} \to \mathit{Val} & \\ \rho \in \mathit{Env} & \mathcal{E}[\![\mathtt{I}]\!]\rho = \mathit{lookup}\ \mathtt{I}\rho & \mathit{lookup} : \mathrm{Id} \to \mathit{Env} \to \mathit{Val} \\ & \mathcal{E}[\![\mathtt{E}_1 \texttt{ + } \mathtt{E}_2]\!]\rho = \mathit{plus}\ \mathcal{E}[\![\mathtt{E}_1]\!]\rho\ \mathcal{E}[\![\mathtt{E}_2]\!]\rho & \mathit{plus} : \mathit{Val} \to \mathit{Val} \to \mathit{Val} \\ & \mathcal{E}[\![\mathtt{E}_1 \texttt{ - } \mathtt{E}_2]\!]\rho = \mathit{minus}\ \mathcal{E}[\![\mathtt{E}_1]\!]\rho\ \mathcal{E}[\![\mathtt{E}_2]\!]\rho & \mathit{minus} : \mathit{Val} \to \mathit{Val} \to \mathit{Val} \end{array}$$

where

$$\begin{array}{ll} \mathit{Val}_{std} = \mathit{Nat} & \mathit{lookup}_{std} = \lambda\,\mathtt{I}.\,\lambda\,\rho.\,\rho\,\mathtt{I} \\ \mathit{Env}_{std} = \mathrm{Id} \to \mathit{Val}_{std} & \mathit{plus}_{std} = \lambda\,v_1.\,\lambda\,v_2.\,v_1 + v_2 \\ & \mathit{minus}_{std} = \lambda\,v_1.\,\lambda\,v_2.\,v_1 - v_2 \end{array}$$

The abstract interpretation is defined by using the same core and giving a different interpretation of the domains and combinators.

If we for example want to do a sign analysis for the arithmetic expressions above, trying to see if we can determine the sign of the expression if we know only the sign of each variable, we could use the following interpretation:

$$\begin{array}{rcl} \mathit{Val} & = & \{p, m\}_\perp^\top \\ \mathit{Env} & = & \mathrm{Id} \to \mathit{Val} \\ \mathit{lookup}_{abs} & = & \lambda\,\mathtt{I}.\,\lambda\,\rho.\,\rho\,\mathtt{I} \\ \mathit{plus}_{abs} & = & \lambda\,v_1.\,\lambda\,v_2.\,(v_1 = p) \to ((v_2 = p) \to p \,\|\, \top) \,\|\, ((v_1 = m) \to ((v_2 = m) \to m \,\|\, \top) \,\|\, \top) \\ \mathit{minus}_{abs} & = & \lambda\,v_1.\,\lambda\,v_2.\,(v_1 = p) \to ((v_2 = m) \to p \,\|\, \top) \,\|\, ((v_1 = m) \to ((v_2 = p) \to m \,\|\, \top) \,\|\, \top) \end{array}$$

Safety can be proven by local reasoning for each combinator instead of a proof involving the entire semantics [JM86a, JM86b, Nie89].

The desired safety condition is expressed as a logical relation R between the abstract and the concrete interpretation of the domains. The relation is defined on the base domains and is extended in the usual way to the compound domains. If we have relations R_A and R_B between domains A_{std} and A_{abs} and B_{std} and B_{abs}, then the relation $R_{A \times B}$ between $A_{std} \times B_{std}$ and $A_{abs} \times B_{abs}$ is given by $R_{A \times B} = \{((a,b),(a',b')) \mid (a,a') \in R_A \wedge (b,b') \in R_B\}$. The relation $R_{A \to B}$ between $A_{std} \to B_{std}$ and $A_{abs} \to B_{abs}$ is given by $R_{A \to B} = \{(f,f') \mid \forall (a,a') : (a,a') \in R_A \Rightarrow (fa, f'a') \in R_B\}$, and so on for the remaining domain constructors.

To prove safety, it is sufficient to prove that the two interpretations are related, *i.e.*, for each combinator of type T, that the T relation R_T holds between the standard and the abstract interpretations of the combinator.

In our example, the obvious relation that we are looking for is

$$R_{Val} = \{(v,v') \mid (v = \bot_{Val} \wedge v' = \bot) \vee (v > 0 \wedge v' = p) \vee (v < 0 \wedge v' = m) \vee v' = \top\}$$

To prove that the sign analysis is safe, it is sufficient to prove that the relation holds between the three combinators, *i.e.*, that $R_{Id \to Env \to Val}(lookup_{std}, lookup_{abs})$, $R_{Val \to Val \to Val}(plus_{std}, plus_{abs})$, and $R_{Val \to Val \to Val}(minus_{std}, minus_{abs})$ hold.

2.2.8 Notation

We mostly use the notation of Schmidt's book on denotational semantics [Sch86].

We use the word *domain* for a CPO, that is, a partially ordered set with a least element and limits of all chains. We will use the symbol $\bot$ to denote the least element, even in cases where it also has another name.

Unless otherwise specified, we assume that a given domain is ordered flatly.

Let A and B be two domains. We will use the following constructors on domains: *Unit* for the one element domain. $A_\bot$ for the lifted domain with a new least element ordered below all the elements from A. $A^\top$ for the domain with a new top elements ordered above all the elements from A. $A \times B$ for the usual product domain ordered pointwise and with $(\bot,\bot)$ as its least element. $A \times B$ for the smash product ordered pointwise and identifying all pairs with a $\bot$ as one least element. A^* for the domain of all finite lists with elements from A with a new least element $\bot_{A^*}$. $A + B$ for the direct sum with no least element. $A + B$ for the coalesced sum, where the least elements of both A and B are merged into one least element. $A \to B$ for the function domain of all Scott-continuous functions from A to B ordered pointwise and with $\lambda\, x.\ \bot$ as the least element. $A \to B$ for the function domain of all strict, Scott-continuous, functions from A to B ordered pointwise and with $\lambda\, x.\ \bot$ as the least element (we work mainly with strict functions since we consider a call-by-value language).

It is well-known that except for $A + B$, these constructs will again be domains [Plo83].

If we have an element v of a product domain $A_1 \times \ldots \times A_n$, then we will use $v \downarrow i$, where $i = 1, \ldots, n$ for the projection onto A_i.

For readability, we will use ordinary parentheses for application and tupling, but square brackets for lists (since we have many pairs of lists). So a pair of lists with respectively m and n elements will be written $([v_1, \ldots, v_m], [w_1, \ldots, w_n])$.

For finite lists with generic element e, we will sometimes use the notation $[\ldots, e_i, \ldots]$, rather than the usual $[e_1, \ldots, e_n]$, since it is shorter. If necessary, the number of elements in the list can be indicated by letting i range from 1 to n : $[\ldots, e_i, \ldots]^{i=1..n}$.

If we have an element v of a domain $A + B$, we will use *cases* v *of* $isA(a) \rightarrow fa \parallel isB(b) \rightarrow gb$ for a strict de-structuring of v, with the meaning that if v is in A, then we apply function f, if v is in B, then we apply function g, and if v is $\perp$, then the result is $\perp$. We will also use the shorthand *let* $A(a) = v$ *in* fa for *cases* v *of* $isA(a) \rightarrow fa \parallel isB(b) \rightarrow \perp$. Conversely, if a is an element of A, we will use $inA(a)$ to inject into $A + B$.

We will use a strict *let*-construct:

$$let\ a = E\ in\ E'$$

when we wish to ensure that E is reduced before E'.

We will use $[I \rightarrowtail v]f$ as a shorthand for $\lambda\ i.\ i = I \rightarrow v \parallel (fi)$ but also as a shorthand for the function that takes a list of pairs and replaces the pair (I, v') with (I, v) or adds (I, v) to the list if there is no (I, v'). (This assumes that there is only one pair in the list for any I and of course preserves that property.)

Let a be a generic element of a domain A. We will denote a generic, i-indexed, chain in A by (a^i) and its limit by $\bigsqcup(a^i)$, implicitly understanding that $i \in \mathbb{N}$.

We will use λ for strict lambda terms and $\boldsymbol{\lambda}$ for non-strict lambda terms.

2.3 Grammars

We will only use very basic properties of grammars here, so we give only the briefest overview.

We will consider a grammar to be two distinct sets of symbols, one that we call terminals and one that we call non-terminals, a set of rules of the form $N \rightarrow r$, where N is a non-terminal and r is a sequence of symbols, and a start-symbol that is also a non-terminal. If we have two rules for the same non-terminal, $N \rightarrow r_1$ and $N \rightarrow r_2$, then we will sometimes use a notational shorthand and write these as one rule: $N \rightarrow r_1 \mid r_2$

We will use the usual language generating function $L(\gamma)$ to denote the language generated by a grammar γ. By abuse of notation we will also write $L(r, \gamma)$ for the language generated by the grammar $S \rightarrow r \cup \gamma$, where S is a non-terminal not used in γ and is the start-symbol of the bigger grammar.

As a slight variation from the usual, though, we consider a language generating function that generates trees rather than strings, similar to the way the well-known BNF-style of grammars generate syntax trees.

We will use the following obvious property of grammars:

Property 1 *If γ is a grammar and N is a non-terminal and r is a right-hand side, then $L(\gamma) \subseteq L(\gamma \cup N \rightarrow r)$.*

In order to do an abstract interpretation, we will want to construct a domain of grammars. An obvious ordering would be by simple set inclusion on the sets of non-terminals, terminals and rules and equality on the start symbol. We use the following obvious property of this ordering:

Property 2 *If we have two grammars, γ and γ', and $\gamma \sqsubseteq \gamma'$, then $L(\gamma) \subseteq L(\gamma')$.*

We also want to take upper bounds in this domain. For this, we use the following property:

Property 3 *If we have two grammars, γ and γ', with the same start symbol, and we consider the grammar γ'' obtained as the union of the sets of terminals and non-terminals and rules in γ and γ', then γ'' is an upper bound of γ and γ'.*

Chapter 3

Predicting Properties of Residual Programs

Partial evaluation as described in the previous chapter is by definition extensionally correct: the residual program is guaranteed to compute the right function. Extensional correctness of residual programs is, of course, a strong argument for using partial evaluation as a general program-development tool.

But to be useful in practice, it is not enough that residual programs are correct. They are also expected to fulfill a number of intensional criteria.

In this chapter, we recall some typical applications of partial evaluation and consider the intensional properties that one would be interested in and could expect to determine in these applications. Such properties can loosely be divided into three classes: those that will always hold, because of the partial evaluation algorithm being used; those that depend only on the partial evaluation algorithm and the source program; and those that also depend on the static data. This division is purely informal and not always clear, but still useful, since the first class of properties may be determined once and for all, given the partial evaluator, while the third class of properties require all the information necessary to generate the residual program, so they are probably most easily discovered by doing just that. The challenging problem is therefore the second class of properties, and whether these properties can be determined automatically.

3.1 Intensional Properties of Partial Evaluation

After the problem of designing a realistic self-applicable partial evaluator for a programming language had been solved with the MIX-project, the intensional properties of the programs involved became a subject of study. This involved the question of which aspects of the MIX partial evaluator could be considered to be essential for self-application (binding-time separation was soon identified, whereas the breadth-first algorithm was implementation-dependent) as well as the structure of the MIX-generated compilers: did they look like "normal" compilers? [JSS85, JSS89, Ses86, BE88, Rom88, GJ89]

Experiments were made to determine the effects on the residual programs of coding changes in the MIX program and the interpreters.

Also, the idea of program specialization had many other potential applications than compiler generation, and with an actual partial evaluator available, several experiments were made, that could, in hindsight, be classified under the title "Can we generate known algorithm X from some other program by partial evaluation?"

3.1.1 Experiments with partial evaluators

A typical example of an experiment with compiler generation by partial evaluation is the MP interpreter that is distributed with the Similix package [Bon91b]. MP (for MiniPascal) is a toy language, imperative, with blocks, commands and expressions over S-expressions. The MP interpreter manipulates an environment and a store (which is implemented as a global data-structure, using the adt facility). As one would expect from compilation, the environment is completely consumed at partial evaluation time, so the target programs generated by partial evaluation access the store (which is still global) directly.

Another typical example is a self-interpreter for the source language of the partial evaluator.

Jones has used this example to set up a condition for the efficiency of a source-to-source partial evaluator: specializing the self-interpreter with respect to itself should give a residual program that (modulo re-naming) is identical to the self-interpreter [Jon90]. This is a simple test of whether the partial evaluator has the effect of removing the interpretive overhead.

A corresponding toy example for higher-order Similix is the Lambda interpreter for a call-by-value lambda calculus with a conditional, constants and an explicit letrec operation.

Except for the generation of compilers for various languages, most of the experiments with application of partial evaluation are probably in pattern matching.

The partial evaluator Redfun [Har77] (which is not self-applicable) was used to compile patterns from a Lisp pattern matcher [Ema80, Ema82].

Dybkjær made some early experiments generating specialized parsers from Earley's general context free parser, but was hindered but the limits of the MIX-system at that time [Dyb85].

Consel and Danvy generate residual programs that have the structure of the Knuth, Morris, Pratt algorithm [KMP77] by taking a naive, two-argument, string matching program and subjecting it to a series of binding-time improvements [CD89]. The KMP-algorithm proceeds by first generating a so-called *next*-table from the pattern string, and then using the table to direct a simple match against the subject string. The structure of the residual programs resembles the *next*-table of the KMP-algorithm, as if the table had been "compiled into" the matcher. This is illustrated by examples of residual programs.

This idea was extended to show that the KMP and the Boyer-Moore algorithms can both be derived from the same algorithm by parametrizing the algorithm on the "first" and "next" functions used to access the strings. Similar techniques can also be used to generate programs with the structure of Weiner-trees [Wei73] in the case where the text is known [Mal89].

The work was further extended to non-linear pattern matching by Danvy [Dan91], using continuation passing style for the binding-time separation.

Jørgensen [Jør91] also compiles patterns by partial evaluation, for the purpose of generating a compiler for a functional language with a pattern matching facility on arguments.

On a related note, Mogensen, Bondorf and Jørgensen derive (residual programs with the structure of) finite automata from regular expressions [Bon90].

The common approach in these and several similar experiments is that a general algorithm is binding-time improved to give a, sometimes curious-looking, but understandable, general source program. The binding-time improvement process is often justified by considering examples of residual programs from the original algorithm, pointing out some redundancy or inefficiency in this program and "reverse engineering" a change in the source program that will eliminate the problem. The binding-time improved program is then specialized with respect to several examples of static data, and the resulting residual programs are seen to have the desired structure, often a structure

similar to some known algorithm. Since the polyvariant specialization algorithm is relatively simple-minded, it is believable to the reader that all residual programs will have the indicated structure. This is not formally proven, though. So the similarity to known data structures is obtained by a mostly automatic transformation, but the similarity is only exemplified, not proven for all possible static data.

3.1.2 Binding-time improvements and binding-time debugging

These experiments did much to improve the understanding of how the polyvariant specialization algorithm works. In particular, it became increasingly clear how the separation of the input into static and dynamic propagates through a source program via both data and control flow. In order to avoid blocking the static flow, source programs must be rewritten to obtain the clearest possible separation between the static actions and the dynamic actions [Bon91b], a process that resembles the "staging" of Jørring and Scherlis [JS86]. This process is known as "binding-time improvement".

Even in the earliest Mix-papers, binding-time improvements were mentioned, and both Malmkjær [Mal89] and Bondorf [Bon91b] summarize heuristics for binding-time improvements. Consel and Danvy advocate a more automatic approach to binding-time improvements using continuation-passing style transformation (CPS) and show how CPS transformation changes the flow of values in the program and thus avoids obstacles to the propagation of static values [CD91].

In order to aid manual binding-time improvements, so-called binding-time debuggers have been developed [CP92, Mos91]. These tools can give the user information about the flow of static and dynamic values during binding-time analysis. The underlying philosophy is that the user of the partial evaluator will have some idea of which parts of the source program will be reduced and which parts will be rebuilt. By specializing an example, one can study the residual program to check whether this seems to hold. If not, one can then use the binding-time debugger to find out why the binding-time analysis classified that particular source term different than expected.

3.1.3 Speed-up analysis

Since the binding-time analysis pre-determines which expressions will be reduced during partial evaluation and which will be rebuilt, it is possible to give a non-trivial estimate of the relative speed-up by partial evaluation.

This aspect has been investigated a little more formally than binding-time improvements. The formalization of the term "speed-up" is in itself not trivial, since we are comparing two programs (source and residual) that take different sets of arguments and compute different functions. Andersen and Gomard prove that with the appropriate definition, the polyvariant specialization algorithm cannot achieve superlinear speed-up, and they design an algorithm that, given a binding-time annotated program, determines the speed-up that can be expected by partial evaluation as an interval in $\mathbb{N} \cup \{\infty\}$.

This research is extended by Jones in [JGS93].

3.2 Intensional Properties of Residual Programs

With the relative success of these experiments in generating realistic residual programs, it looked like partial evaluation could become a practical tool in program development. To verify this, people have started to consider the quality of residual programs in general, not just the quality of single, experimental, residual programs.

This involves finding general heuristics for binding-time improvements and the question of which "good" properties we can ensure that the residual programs will have.

For example when partial evaluation is used for compilation, we have one specific interpreter that we intend to specialize with respect to different source language programs. This gives us an infinite set of compiled programs (residual programs).

In order to find out if the compilation is adequate, we would like to determine some properties of the compiled programs. This includes some estimates about size and complexity, but also structural properties, such as whether all administrative (or "compile-time") operations have been eliminated from compiled programs, whether they use data-structures and storage in an appropriate way (*e.g.*, no duplication of resources that are single-threaded in the interpreter), whether all procedure calls are tail-recursive, and (in the case of a partial evaluator with the same source and target language) whether the residual programs are written in the same, or possibly a smaller, subset of the language as the interpreter.

If we can show, for example, that the target program generated by partially evaluating the interpreter is guaranteed to be within a sufficiently restricted subset of the residual language (for example, that all calls are tail-calls), then we can perform a one-to-one translation to low-level code, in the spirit of Wand or Fradet and LeMetayer [Wan82, FL91].

If we imagine a partial evaluator for a lazy language, one could imagine that if we could establish that a particular term destructing an argument is always present in the residual programs, then that argument can safely be considered strict. Conversely, if one knew that a particular parameter was not strict in the source program, it might be possible to establish that its instances in the residual program were not strict, either.

We are mainly interested in the structure of the residual programs in order to show formally that the various algorithms obtained by partial evaluation of pattern matchers do indeed correspond to known structures and algorithms

In order to show that they do indeed compile patterns effectively, we are also interested in their operations on the string and in the relative speed-up.

In order to show, for example, that automata are generated from a regular expression matcher, we need to show that the residual programs have procedures or expressions corresponding to the states and transitions of the automata.

In general, we are concerned about properties such as

- The size of the residual program. This includes both absolute size and the size of the code generated at each residual program point.
- The run-time of the residual program. Again, this includes both the usual run-time complexity of the entire program and the more local reasoning about how much computation is residualized at each program point, compared to how much is executed at partial evaluation time (the speed-up by partial evaluation).
- The space-complexity of the residual program, again both absolute and relative to the source program.
- The call-structure of the residual program, are there loops and which residual procedures call each other.
- The use of data-structures in the residual program.

- Which syntactic forms and language constructs the residual program uses and which combinations they appear in.
- Whether all source terms that are intended to be static have been eliminated.

Let us consider what these properties depend on.

Some of the properties of residual programs are entirely determined by the partial evaluator. The proto-typical example is the programming language in which the residual programs are written. The partial evaluator generates programs in a specific language. Furthermore, it will often not use all the constructs of that language, for internal simplicity, and this subset can be determined simply by examining the partial evaluator.

But it is also possible to determine more subtle properties.

As an example of this, it has been proven for the Similix specializer that source computations are not duplicated in the residual program. This is interesting since, without this property, it is easy to construct examples where the residual program is slower than the source program on the same input.

Some properties of residual programs clearly depend on both the source program and the static data, as well as the partial evaluator being used. As an example, the residual programs of the KMP-example [CD89] will have recursive calls if and only if there is a repeated character in the pattern. More general examples include the absolute of the residual programs, which is typically a function of the static data.

Most of the intensional properties of the residual programs, however, depend on the source program and the division of the arguments into static and dynamic, and only to a minor extent on the actual static data.

In other words, residual programs stemming from the same source program can be expected to have many things in common. Because of the polyvariant specialization, each procedure in the residual program will be a specialized instance of a procedure in the source program. Because of the binding-time annotations, much of the text of the specialized instance can be derived from the source program. Also, procedure calls in the residual program will be specialized instances of calls in the source program, so it should be possible to determine the corresponding source procedure, but not which instance, since that may depend on the static data. So a "framework" of the call structure of the residual programs can be derived from the call structure of the source program.

3.3 Detecting Properties of Residual Programs Automatically

Consider the intermediate case, where we assume that the source program and the binding-time pattern are known, but the static data is unknown or only an abstraction of it is given. This defines an infinite family of residual programs.

In the previous section, we described a number of properties that depend only on this, that is, are common to such a family, and how we might take advantage of knowing such properties in advance.

How could we automatically determine properties of the programs in this family? That is, could we for example give a finite description of this family, from which we could derive information about call structures, use of language constructs and data structures, *etc.*

We want to approach this problem in a formal manner and produce provably correct information about the intensional properties of residual programs.

Although existing partial evaluators become increasingly complex, the basic algorithm of polyvariant specialization [Bul84] is characterized by its regularity. Based on a small number of parameters it performs one of a small number of actions, over and over. As output it produces a collection of chunks of the source program that were determined by the binding-time analysis and are glued together in a structure determined from the processing of the static data.

It is this knowledge of binding-time based regularity that we use to rewrite source programs to specialize well: write the code just so and this piece will always be eliminated – that piece always reconstructed. So it is not unreasonable to expect that given the source program it might be possible to automatically determine intensional properties that will hold for all residual programs that can be produced by specializing this source program.

3.3.1 Examples

For a very small example, consider the continuation-passing exponentiation program in figure 3.1. If we specialize this program with respect to the exponent 0, we get the residual program to the left in figure 3.2. If we specialize it with respect to the exponent 5, we get the program to the right.

This is of course not surprising. However, we would like to be able to say with certainty that the result of specializing the exponent will *always* be of the form `(define (exp arg) body)`, where `body` is either `1` or `(* arg body)`.

Since there is no upper bound on the number of multiplications, we cannot expect to describe this family of residual programs by ennumerating the members of the set of all possible programs. Instead, we need to find a finite representation of this infinite family of programs.

```
(define (exp x n)
    (exp-a x n (lambda (v) v)))

(define (exp-a x n k)
    (if (zero? n)
        (k 1)
        (exp-a x (sub1 n) (lambda (v) (* x (k v))))))
```

Figure 3.1: A continuation-passing exponentiation program

```
(define (exp-0 x_0)
    1)
```

```
(define (exp-0 x_0)
    (* x_0 (* x_0 (* x_0 (* x_0 (* x_0 1))))))
```

Figure 3.2: The results of specializing the exponent program with respect to 0 and 5 as the exponent

For another example, consider the following fragment of the MP interpreter, mentioned above:

```
(define (evalExpression E env)
   (cond
      [(isConstant? E) (constant-value E)]
      [(isVariable? E) (lookup-store (lookup-env (E->V E) env))]
      [(isPrim? E)
       (let ([op (E->operator E)])
          (cond
             [(is-cons? op) (cons (evalExpression (E->E1 E) env)
                                  (evalExpression (E->E2 E) env))]
             [(is-equal? op) (equal? (evalExpression (E->E1 E) env)
                                     (evalExpression (E->E2 E) env))]
             [(is-car? op) (car (evalExpression (E->E E) env))]
             [(is-cdr? op) (cdr (evalExpression (E->E E) env))]
             [(is-atom? op) (atom? (evalExpression (E->E E) env))]
             [else "Unknown operator"]))]
      [else "Unknown expression form"]))
```

When we specialize the MP interpreter with respect to a program, the specialized pieces of this fragment appear as `(lookup-store 3)`, `(cons 1 (lookup-store 4))`, and so on. This matches with our expectation that the store is eliminated.

We would like to establish that all residual fragments would have this form, that is, that the environment variable is never accessed. If that was the case, we could remove the definitions in the adt-file that relates to the environment, and still be sure that we could run the target programs. Also, we could be sure that all environment lookup had been done at partial evaluation time (aka compile time).

3.3.2 Towards an analysis

To find out how to determine properties of residual programs from the source program and the partial evaluator, let us consider our goals

- we are interested in the *output* of the partial evaluator, that is, not in its internal operations, except where this will somehow influence the output. This differs from compiler-oriented analyses, 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, for example, whether a number is odd or even. What is important, however, is for example whether a structured value is a residual expression obtained by applying the primitive `_sim-build-cond`.

A purely syntactic analysis of the partial evaluator would not be enough for our purposes. It could tell which syntactic terms the partial evaluator could generate, but not how they fit into each other. This could be determined with some kind of flow analysis.

Most program analyses concern the runtime behaviour of the program, though some, particularly the toy examples, such as sign or odd/even, concern static information about the result. Here we are only interested in the output of the program: its text, its structure, possibly its run-time behaviour. And we only have one program that we want to analyze: the partial evaluator.

To investigate the feasibility of such an analysis, let us for the moment assume that a partial evaluator takes two arguments, a binding-time analyzed 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 p is p itself (the actual source program is given) and the abstraction of the static data

is T (nothing is known about the static data), 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 it seems unnecessary to design an analysis for an entire language, when we only want to analyze one program: the partial evaluator.

We could shift our focus, though, and claim that the program we want to analyze is not the partial evaluator, but the binding-time annotated source program. This would mean that the partial evaluator was in fact the standard interpretation of the program, and we would be looking for an abstract interpretation of this: an "abstract partial evaluation".[1] Safety of such an abstract interpretation would have to be proven relative to the partial evaluator. Although partial evaluators have many features in common with interpreters, they are generally more complex, both because the language they define has twice as many syntactic features as the one of a corresponding interpreter, and because of the extra administration of the polyvariance. If we want to prove our analysis safe, this might not be a good strategy.

So we recall that there is actually another program that produces the same output as the partial evaluator with less computation: the generating extension.

When we have a source program, p, the generating extension of p, G_p, takes the static data as an argument and produces the specialized version of p, with respect to that data, as output. In other words, G_p knows how to specialize p and no other program. So a non-standard interpretation of G_p, where the abstraction of the static data is "nothing is known", gives an abstract description of the possible specialized versions of p.

This way, safety of the abstract interpretation can be proven relative to the standard semantics of the relatively simple residual language, instead of with respect to the entire partial evaluator.

To see that this is also a practical approach, we note that intuitively a generating extension produced by a polyvariant specializer generates a residual program by recursively composing delimited contexts. The delimited contexts are determined by the binding-time analysis. This can clearly be seen in examples, though they do not give the whole picture. The piece missing is essentially the principle for generation of residual procedure definitions when calls are encountered.

An abstract description giving the delimited contexts and some information on how they fit into each other should establish some of the intensional properties that we are interested in.

3.3.3 Example: the generating extension as a syntax constructor

Let us consider a very simple example, to show what we mean by saying that a polyvariant specializer composes delimited pieces of syntax and to outline the basic features of the proposed abstraction.

The source program given to the left in figure 3.3 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 (we have already seen this program in section 2.1.3).

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 at the top right in figure 3.3. The two calls to append have been unfolded during specialization and the static lists have been broken down into their components.

If we specialize it with respect to () and (c d e), we get the second result to the right in figure 3.3. Not surprisingly, this program looks pretty much like the first one. Again the calls to append have been unfolded and the static lists decomposed.

[1] This would relate to the ideas of AMIX [Hol88], where the partial evaluator is seen as a non-standard semantics for the annotated program. The parametrized partial evaluation of Consel and Khoo [CK91] also investigate the idea of a partial evaluation semantics as a way of assigning meaning to an annotated program.

```
(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)))))

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

Figure 3.3: The source program main (left) and the versions specialized with respect to (a b) and (c d) and with respect to () and (c d e) using Similix (right)

```
(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))))
```

Figure 3.4: Binding-time annotated version of the source program in figure 3.3

In fact 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 cons'ing some constant values onto the argument and the second of which is the result of cons'ing some other constant values onto the argument.

Similix obtains this result by first binding-time analyzing the source program with respect to the two first arguments being static and the third being dynamic. The binding-time analysis determines which parts of the computation depends only on the values of x and y, so they can be executed, and which parts depend also on the value of z, that is, they have to be rebuilt. The binding-time analysis of the source program in figure 3.3 gives the annotated program in figure 3.4. We use the convention that static expressions are in italics and dynamic expressions are in boldface. The identifier append is annotated as static since the calls will be unfolded. This is in fact only an approximation of the actual annotations in Similix, but the difference will not influence this example.

If we consider only the dynamic parts of the annotated program, we see that the residual program will be built from the definition of a procedure with one argument. The body of this procedure will cons two things. These things will be either the argument or the cons of a constant and another "thing".

This description actually fits what happens in the generating extensions. Unfortunately generating extensions are usually too large and generally incomprehensible to show[2], but in figure 3.5 we give a simplified pseudo-code version of the generating extension produced by Similix for the source program in figure 3.3 (most importantly, we have removed all references to the seenbefore-list and

[2]This is both due to limitations in the current partial evaluation techniques and to the fact that names in a generating extension are derived from names in the specializer and thus do not correspond to the purpose of the procedures and variables being named.

```
(define (specialize-0 value*_0)
    (let* ([value_1 (list-ref value*_0 0)]
           [value_2 (list-ref value*_0 1)]
           [p_5 (_sim-build-var (_sim-generate-param-name 'z -1))]
           [residual-name
            (_sim-generate-proc-name! 'main (cons value_1 (cons value_2 (cons p_5 ()))))])
      (_sim-build-def residual-name_12
                      p_5
                      (_sim-build-primop2 'cons
                                          (_sim-process-expr p_5 value_1)
                                          (_sim-process-expr p_5 value_2)))))
(define (_sim-process-expr-0-2 r_0 r_1)
    (if (null? r_1)
        r_0
        (_sim-build-primop2 'cons
                            (_sim-build-cst (car r_1))
                            (_sim-process-expr-0-2 r_0 (cdr r_1)))))
```

Figure 3.5: "Pseudo-Similix" generating extension for the program in figure 3.3

```
specialize-0   ::=   (define (ID-main ID-z)
                       (cons expr expr))
expr           ::=   ID-z | (cons Cst expr)
```

Figure 3.6: BNF describing the possible results of specializing the program of figure 3.3 with the two first arguments static and a dynamic third argument

polyvariance).

Its output is an infinite family of residual programs, since they are constructed recursively. But since the family is regular, we can describe it by a grammar.

The BNF of figure 3.6 can be obtained from the generating extension in figure 3.5 by tracing the constructions the generating extension performs.

Of course in this reasoning we have used the fact that the "things" are really the results of static calls to append and will be inserted in the places where the calls occur. Also we use that a static sub-expression of a dynamic expression will appear as a constant in the residual program.

3.3.4 Abstract representation of the family of residual programs

The example above suggests that we can determine quite a lot of generic information about the family of residual programs by abstract interpretation of the generating extension of the source program.

The abstract interpretation should give information about output, that is, the key ingredient is an appropriate abstraction of values and data-structures handled by the program. In particular, it should determine any syntactic constructions and constants appearing at regular places in the syntax.

As suggested by, for example, Mogensen, this can be obtained with a grammar representation. A set of data-structures can be represented (safely, but not necessarily accurately) by a grammar that defines a superset of the set. Grammars also seem to be a particularly apt representation in this case, since BNF-style grammars are customarily used to describe program syntax.

The symbols of the residual syntax – parentheses, operator names, tags such as `cst` or `primopt` – will be the terminals of the grammars. The procedure names of the generating extension will be the non-terminals. To represent the results of primitive operators such as the unique name generators, which we know will be a string containing its argument string[3], we can introduce another family of non-terminals. Similarly, we can introduce non-terminals to represent the sets of ordinary values in the program: booleans, numbers, strings, *etc.*

Since we are mainly interested in the residual syntax, we are in a rather fortunate situation: there are only constructors on this data-type in Similix, no destructors or predicates. If we abstract all other primitives in a trivial way, this greatly simplifies our reasoning about termination and complexity of the analysis.

The grammars can be partially ordered by subset inclusion on the rules. Then we have that if two grammars are ordered, then the sets they define must be ordered (by the usual subset ordering), while the opposite may not be true.

Termination of the abstract interpretation is guaranteed by observing that for any given program, we have a finite set of terminals and non-terminals, and a finite length of the possible right-hand sides, giving only finitely many possible grammars for that program.

3.3.5 Summary: how do we represent families of residual programs and how do we obtain such representations from source programs

We use a depth-first algorithm, that is, when we encounter a procedure call, we symbolically evaluate the call before proceeding with the treatment of the context of the call. This approach has the advantage that the abstract interpretation is very similar to the standard interpretation, since there is no global fixed point. The close alignment between the two interpretations will simplify a safety proof based on a "two-level" formulation. The disadvantage is that we need to introduce local fixed points in order to prove the algorithm safe.

Outline of the algorithm: How to add rules to the grammar

Following Jones we want to generate a non-terminal (*ie.*, a grammar rule) for each procedure in the program. The right hand side of the rule describes the result of that procedure. 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.

Practically this means that when a procedure is applied, the analysis should determine the abstract value of the body of the procedure. The resulting abstract value is the right hand side corresponding to that procedure. The rule with the non-terminal of the procedure and that right hand side is added to the grammar.

In the extended grammar, the non-terminal of the procedure now denotes the result of the procedure. Thus it is sufficient to replace the procedure by a constant function returning the non-terminal: if the procedure contains a recursive call, we do not need to execute it, since we already know that the result of the procedure is its corresponding non-terminal. So inside the procedure (and after evaluation as well), we can use a memo-function to remind us that this procedure has already been evaluated and its value is its non-terminal. This memo-function is naturally a part of

[3] A property that is easily established from their definition in the adt-file.

the procedure environment, so when we apply a procedure, we first look it up in the memo-function, to see if we already know its result.

So the abstract value of the body can be determined by symbolically evaluating the body in an environment with a memo-function where the procedure denotes a constant function that always returns the corresponding non-terminal (thus ensuring termination). This gives the result of applying the procedure: a right hand side, consisting of the non-terminal corresponding to the procedure, and the extended grammar.

Note, however, that a non-terminal alone denotes the empty language. So for the safety relation to hold, it is not enough to replace the procedure with a constant function returning the corresponding non-terminal. The constant function should also return a grammar with a rule for that nonterminal. Clearly the right hand side of the non-terminal in that grammar should be the result of the symbolic evaluation of the body. But inside the procedure, the right hand side of this (expected) rule is exactly what we are trying to compute! This is solved by introducing a fixed point over the result of applying the procedure.

So an abstract value has to be both a term (a potential right hand side) and a grammar defining the non-terminals used in the right hand side term.

Adding the residual call structure to the grammar

While the method outlined above does give a grammar representation of the output of generating extensions, *i.e.*, of residual 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 residual programs.

Obtaining this is actually relatively simple in Similix. By modifying the primitives to produce a non-terminal instead of a procedure name at a call site and a grammar rule instead of a definition, the structure of the grammar will also reflect the call-structure of the generated programs. To claim that this kind of grammars also defines a family of residual programs, we have to modify the function generating a language from a grammar, since a non-terminal in a call is now denotes a procedure name, while the same non-terminal in a program listing denotes a residual procedure definition. But since it corresponds to the generation of both a residual call and a residual definition in the generating extension, we know that the use of the non-terminal does indeed reflect the call-structure of the residual program.

3.4 Conclusion

We propose to determine properties of residual programs by analyzing generating extensions. The analysis produces a grammar with non-terminals corresponding to the procedures of the generating extension and to the procedures of the residual program that are instances of the same source procedure. We have shown some examples, indicating that such a representation does capture structural aspects of the residual programs and that this method will produce reasonably informative descriptions of the residual programs.

In the next chapter we give a precise formulation of the grammar-generating analysis as a non-standard interpretation, first for first-order Similix and then an extension to higher-order Similix.

Chapter 4

Grammar-Generating Analysis

In this chapter we consider how to represent and determine the properties outlined in the previous chapter. These considerations lead to the design of an abstract interpretation using grammars to represent values. We give a detailed version of such an analysis for first-order Similix. This is then extended to higher-order by integrating a closure analysis into the grammar-based analysis.

Before we proceed with the design of the analysis, however, we will describe more precisely the source language that we are concerned with.

4.1 The Source Language

We use the Similix specializer and its source language. In this section we give the abstract syntax and a standard denotational semantics for this language. We emphasize the aspects of the language that are used in the specializer itself.

The Similix source language is essentially a subset of Scheme. It is untyped, call-by-value, and has limited notions of store and side-effects and of abstract data-types. It has a Scheme syntax.

As described in section 2.1.7, the Similix notion of abstract data-types allows the user to extend the recognized set of primitive operations. The set of primitives declared for the specializer includes operations for constructing residual syntax and operations for declaring and manipulating two global structures, containing the list of specialization points that have already been encountered (the "seenbefore" list) and the resulting specialized procedures (the "out" list). We will treat these primitives and the terms they operate on as language primitives rather than defining them in terms of their Scheme definitions. Thus we also give the standard semantics for these primitives.

Because we first develop an analysis for first-order Similix and then extend it to higher-order, we will give separate semantics for first-order Similix and higher-order Similix.

4.1.1 Standard semantics of the first-order language

Syntax

The BNF of the first-order source language is given in figure 4.1 (there are a few more constructs handled in Similix, but we will ignore these here, partly because most are syntactic sugar).

Domains

The domains of the standard semantics are shown in figure 4.2.

P ∈ Pgm	C ∈ Cst	O ∈ Op
E ∈ Expr	I ∈ Id	F ∈ Proc-Name

P ::= ((define (F_1 I_{1_1} ... I_{n_1}) E_1) ... (define (F_m I_{1_m} ... I_{n_m}) E_m))

E ::= C | I | (if E_0 E_1 E_2) | (O E_1 ... E_n) | (F E_1 ... E_n) | (let ((I E_1)) E_2)

O ::= cons | car | cdr | ... | _sim-build-def | _sim-build-cond | _sim-generate-proc-name! ...

Figure 4.1: BNF for the first-order source language

$v \in Val = Nat + Bool + Str + Pair + Res\text{-}Expr$	$f \in Proc = Val^* \rightarrow Store \rightarrow Result$
$(v_1, v_2) \in Pair = Val \times Val$	$\sigma \in Store = Val \times Val$
$\rho \in Env = (\text{Id} \rightarrow Val)_\perp$	$(v, \sigma) \in Result = Val \times Store$
$\varphi \in Penv = (\text{Proc-Name} \rightarrow Proc)_\perp$	

Figure 4.2: Standard semantics of the first-order language, domains

The domain of values is a coalesced sum of the flatly ordered domains of naturals, booleans, strings, *etc.* Note that this domain also contains syntax trees describing residual programs (*Res-Expr*), since Similix has primitives for handling such structures.

Thus for example we have a Similix primitive named `_sim-build-primop-1`, that takes two arguments, a unary primitive and a syntax tree and builds the syntax tree where the unary primitive is applied to the syntax tree argument. So if the first argument is the constant `'add1` and the second argument is the residual expression `(cst 1)`, then the result is the residual expression `(primop1 add1 (cst 1))`, which is an element of *Res-Expr*.

RPgm	::=	add-residual-definition RDef RPgm | ϵ
RDef	::=	definition I I^* RExpr
RExpr	::=	cst C | var I | cond $RExpr_1$ $RExpr_2$ $RExpr_3$ | let I RExpr RBody | begin $Rexpr^*$
	|	primop0 O | primop1 O $RExpr_1$ | primop2 O $RExpr_1$ $RExpr_2$
	|	primop3 O $RExpr_1$ $RExpr_2$ $RExpr_3$ | primop4 O $RExpr_1$ $RExpr_2$ $RExpr_3$ $RExpr_4$
	|	$primop^*$ O $RExpr^*$ | pcall F $RExpr^*$

Figure 4.3: A BNF corresponding to the residual syntax constructors of Similix

So *Res-Expr* is in fact a collection of syntax trees, that is, the term algebra corresponding to the BNF describing the residual syntax. Note that the algebra is only used as a tool to describe the elements of the domain. We order the residual expressions as a flat domain. Figure 4.3 gives the BNF corresponding to the Similix residual syntax constructors. Note that for convenience, we have only one domain for all the residual syntactic categories.

The store contains only the two predefined global structures. This is reflected in the semantics by defining the *Store* domain to be $Val \times Val$.

In order to simplify the treatment, we have separated the environment into a value environment and a procedure environment. Procedures are functions taking a list of values and a store into a

result. A result is a value and a store, that is, three values.

The domain *Penv* is a domain of functions from procedure names to procedures.

Valuation functions

$\mathcal{P}$: Pgm → Proc-Name → *Val** → *Val*

$\mathcal{P}[\![$(define (F$_1$ I$_{1_1}$ ... I$_{n_1}$) E$_1$) ... (define (F$_m$ I$_{1_m}$ ... I$_{n_m}$) E$_m$))$]\!]$ F v^* =

let $\varphi = fix\,\lambda\,\varphi.\,[\mathtt{F}_1 \mapsto \lambda\,[v_1,\ldots,v_{n_1}]\,\sigma.\,\mathcal{E}[\![\mathtt{E}_1]\!][\mathtt{I}_{1_1} \mapsto v_1,\ldots,\mathtt{I}_{n_1} \mapsto v_{n_1}]\lambda\,i.\,\perp_{Val}\varphi\sigma,\ldots,$
$\mathtt{F}_m \mapsto \lambda\,[v_1,\ldots,v_{n_m}]\,\sigma.\,\mathcal{E}[\![\mathtt{E}_m]\!][\mathtt{I}_{1_m} \mapsto v_1,\ldots,\mathtt{I}_{n_m} \mapsto v_{n_m}]\lambda\,i.\,\perp_{Val}\varphi\sigma]\lambda\,f.\,\perp_{Proc}$
in $((\varphi\,\mathtt{F}\,v^*\,\sigma_{init})\downarrow 1)$

$\mathcal{E}$: Expr → *Env* → *Penv* → *Store* → *Result*

$\mathcal{E}[\![\mathtt{C}]\!]\rho\varphi\sigma = (\mathcal{C}[\![\mathtt{C}]\!],\sigma)$

$\mathcal{E}[\![\mathtt{I}]\!]\rho\varphi\sigma = (\rho\,\mathtt{I},\sigma)$

$\mathcal{E}[\![$(if E$_0$ E$_1$ E$_2$)$]\!]\rho\varphi\sigma$ = *let* $(Bool(b),\sigma_0) = \mathcal{E}[\![\mathtt{E}_0]\!]\rho\varphi\sigma$ *in* $(b \to \mathcal{E}[\![\mathtt{E}_1]\!]\rho\varphi\sigma_0 \,[\!]\, \mathcal{E}[\![\mathtt{E}_2]\!]\rho\varphi\sigma_0)$

$\mathcal{E}[\![$(O E$_1$ ... E$_n$)$]\!]\rho\varphi\sigma$ = *let* $(v_1,\sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\rho\varphi\sigma$
in ... *let* $(v_n,\sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\rho\varphi\sigma_{n-1}$
in $\mathcal{O}[\![\mathtt{O}]\!]\,[v_1,\ldots,v_n]\,\sigma_n$

$\mathcal{E}[\![$(F E$_1$ ... E$_n$)$]\!]\rho\varphi\sigma$ = *let* $f = (\varphi\,\mathtt{F})$
in *let* $(v_1,\sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\rho\varphi\sigma$
in ... *let* $(v_n,\sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\rho\varphi\sigma_{n-1}$ *in* $f\,[v_1,\ldots,v_n]\,\sigma_n$

$\mathcal{E}[\![$(let ((I E$_1$)) E$_2$)$]\!]\rho\varphi\sigma$ = *let* $(v,\sigma') = \mathcal{E}[\![\mathtt{E}_1]\!]\rho\varphi\sigma$ *in* $\mathcal{E}[\![\mathtt{E}_2]\!]\,[\mathtt{I} \mapsto v]\rho\varphi\sigma'$

$\mathcal{C}$: Cst → *Val*

$\mathcal{O}$: Op → *Val** → *Store* → *Result*

$\mathcal{O}[\![$cons$]\!]\,[v_1,v_2]\,\sigma = (in\,Pair(v_1,v_2),\sigma)$

$\mathcal{O}[\![$car$]\!]\,[v]\,\sigma = (let\ Pair(v_1,v_2) = v\ in\ v_1,\sigma)$

...

$\mathcal{O}[\![$_sim-build-def$]\!]\,[v_1,v_2,v_3]\,\sigma = (in\,Res\text{-}Expr(definition,v_1,v_2,v_3),\sigma)$

$\mathcal{O}[\![$_sim-build-cond$]\!]\,[v_1,v_2,v_3]\,\sigma = (in\,Res\text{-}Expr(cond,v_1,v_2,v_3),\sigma)$

...

$\mathcal{O}[\![$_sim-generate-proc-name!$]\!]\,[v_1,v_2]\,(\sigma_1,\sigma_2)$ =
cases $(seenb4?\,v_1\,v_2\,\sigma_1)$ *of* $isVal(v_n) \to (in\,Pair(v_n,True)),(\sigma_1,\sigma_2))$
$[\!]\ isUnit() \to$ *let* $v_n = gen\text{-}new\text{-}name\,v_1\,v_2\,\sigma_1$
in $(in\,Pair(v_n,False),([v_1,v_2,v_n] :: \sigma_1,\sigma_2))$

where

seenb4? : *Val* → *Val* → *Val* → $(Val + Unit)_\perp$

Figure 4.4: Standard semantics of the first-order language

The valuation functions of the standard semantics are given in figure 4.4.

Since we will only be interested in the legal output of programs, the standard behaviour on errors is irrelevant to our concerns. So for readability, we have simplified the error treatment, so an error simply gives $\perp$.

The procedure environment is built by binding each of the function names to a function defined using the procedure environment, since all the procedures are mutually recursively defined. This is of course accomplished by taking the fixed point over the procedure environment.

Note that the semantics specifies a left to right evaluation order of arguments. This corresponds to Similix, but deviates from the usual Scheme semantics, that leaves the evaluation order of arguments explicitly unspecified.

We have the usual strict conditional, that determines the value of the test and selects one of the alternates.

We only give samples of the the definition of $\mathcal{O}$ and we omit $\mathcal{C}$, that translates syntactic constants into values.

_sim-generate-proc-name! is the name of one of the dedicated primitives used in the Similix specializer. When a procedure application is to be generated by the specializer in the standard semantics, **_sim-generate-proc-name!** is called to produce the residual name of the procedure. However, since the source procedure might already have been specialized with respect to these particular static arguments, the primitive checks the source name and the static values to see if a name has already been generated. If this is the case, it simply returns that name, otherwise it generates a new name. It also returns a flag telling whether the name is new. This flag is used in the specializer to determine whether to specialize the source procedure and add its residual definition to the list of results (by side-effect) before completing the generation of the application.

The auxiliary *seenb4?* is a lookup function that returns v_n if there is a triple (v_1, v_2, v_n) in σ_1 and *()* otherwise.

Note that the semantics actually uses a single-threaded counter in the store to generate a new, unique, name. For simplicity we have totally omitted this from the semantics, and we simply assume that we can, by magic, produce new, unique, strings to use as names.

4.1.2 Standard semantics of the higher-order language

In this section we give a standard denotational semantics of the higher-order source language of Similix.

The semantics is closure-based and uses a globalized procedure environment[1]. It is relatively simple to prove this version equivalent to a more usual closure-semantics, for a given program with unique program points, so we omit this step here.

Syntax

The BNF of the higher-order source language is given in figure 4.5. Note that we no longer have a separate syntactic category of procedure names.

```
P ::= (define (I0 I1₀ ... In₀) E0) ... (define (Im I1ₘ ... Inₘ) Em)
E ::= C | I | (if E0 E1 E2) | (lambda (I1 ... In) E) | (O E1 ... En)
    | (E0 E1 ... En) | (let ((I E1)) E2)
O ::= cons | car | cdr | + | ...
    | _sim-build-def | _sim-build-cond | _sim-generate-proc-name!  ...
```

Figure 4.5: BNF for the higher-order source language

For simplicity, we have assumed that closures can only be applied or passed to procedures (possibly other closures) or returned from calls. We do not allow other operations on closures, such as cons'ing a closure onto a list.

[1] This will simplify the alignment with the abstract interpretation in the next chapter.

$$
\begin{aligned}
&\texttt{P} ::= \texttt{(define (I}_0\ \texttt{I}_{1_0} \ldots \texttt{I}_{n_0}\texttt{)}\ \texttt{E}_0^{I_0}\texttt{)} \ldots \texttt{(define (I}_{1_m}\ \texttt{I}_1 \ldots \texttt{I}_{n_m}\texttt{)}\ \texttt{E}_m^{I_m}\texttt{)}\\
&\texttt{E}^p ::= \texttt{C} \mid \texttt{I} \mid \texttt{(if}\ \texttt{E}_0^{p.0}\ \texttt{E}_1^{p.1}\ \texttt{E}_2^{p.2}\texttt{)} \mid \texttt{(lambda (I}_1 \ldots \texttt{I}_n\texttt{)}\ \texttt{E}^{p.2}\texttt{)} \mid \texttt{(O}\ \texttt{E}_1^{p.1} \ldots \texttt{E}_n^{p.n}\texttt{)}\\
&\qquad \mid \texttt{(E}_0^{p.0}\ \texttt{E}_1^{p.1} \ldots \texttt{E}_n^{p.n}\texttt{)} \mid \texttt{(let ((I}\ \texttt{E}_1^{p.1}\texttt{))}\ \texttt{E}_2^{p.2}\texttt{)}\\
&\texttt{O} ::= \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{+} \mid \ldots\\
&\qquad \mid \texttt{\_sim-build-def} \mid \texttt{\_sim-build-cond} \mid \texttt{\_sim-generate-proc-name!} \quad \ldots
\end{aligned}
$$

Figure 4.6: BNF for the higher-order source language with program points as inherited attributes

In order to get unique references to the lambda abstractions in the program, we decorate the syntax with program points that we consider to be attributes of the internal nodes of the syntax tree. A BNF of the syntax decorated with program points is given in figure 4.6. We base ourselves on the observation that in a given program we have a function from the program points of lambdas to the text of lambda abstractions. This gives us a function, in the semantics, from program points to the first component of the denotation of lambda abstractions (the procedure templates), since we are using a closure semantics where the context environment of a lambda is specified separately.

Domains

$$
\begin{aligned}
&v \in \mathit{Val} = \mathit{GroundVal} + \mathit{Closure}\\
&g \in \mathit{GroundVal} = \mathit{Nat} + \mathit{Bool} + \mathit{Str} + \mathit{Pair} + \mathit{Res\text{-}Expr}\\
&(v_1, v_2) \in \mathit{Pair} = \mathit{GroundVal} \times \mathit{GroundVal}\\
&c \in \mathit{Closure} = \mathit{Program\text{-}Point} \times \mathit{Env}\\
&\rho \in \mathit{Env} = (\mathrm{Id} \rightarrow \mathit{Val})_\perp\\
&\Phi \in \mathit{Proc\text{-}Env} = \mathit{Program\text{-}Point} \rightarrow \mathit{Proc\text{-}Template}\\
&\pi \in \mathit{Proc\text{-}Template} = \mathit{Env} \rightarrow \mathit{Proc}\\
&\varphi \in \mathit{Proc} = \mathit{Val}^* \rightarrow \mathit{Proc\text{-}Env} \rightarrow \mathit{Store} \rightarrow \mathit{Result}\\
&\sigma \in \mathit{Store} = \mathit{Val} \times \mathit{Val}\\
&(v, \sigma) \in \mathit{Result} = \mathit{Val} \times \mathit{Store}
\end{aligned}
$$

Figure 4.7: Standard semantics of the higher-order language, domains

The standard domains for the semantics of the higher-order language is given in figure 4.7.

The domain of values is now the sum of ground values and closures, representing the higher-order values.

A procedure template is a function from environments to procedures. The intention of this is that the environment supplies the definitions of any free identifiers in the procedure template.

Since we have globalized the procedure templates, based on the program points, a closure is simply a program point and an environment. As for the first-order language, we have included integers, booleans, strings, pairs, and residual expressions (output syntax) as ground values.

Since there is no natural purpose for a least element in *Program-Point*, we will simply assume that *Program-Point* is a discreetely ordered set. We note that $\mathit{Program\text{-}Point} \times \mathit{Env}$ and $\mathit{Program\text{-}Point} \rightarrow \mathit{Proc\text{-}Template}$ are still domains.

Valuation functions

$$\mathcal{P} : \text{Pgm} \to \text{Id} \to \mathit{Val}^* \to \mathit{Result}$$

$$\mathcal{P}[\![\texttt{(define (}\mathtt{I}_0\ \mathtt{I}_{1_0}\ \ldots\ \mathtt{I}_{n_0}\texttt{)}\ \mathtt{E}_0^{\mathtt{I}_0}\texttt{) ... (define (}\mathtt{I}_m\ \mathtt{I}_{1_m}\ \ldots\ \mathtt{I}_{n_m}\texttt{)}\ \mathtt{E}_0^{\mathtt{I}_m}\texttt{)}]\!]\,\mathtt{I}\,v^* =$$
$$\mathit{let}\ \Phi_0 = \mathcal{L}[\![\mathtt{E}_0^{\mathtt{I}_0}]\!]([\mathtt{I}_0 \mapsto \lambda\,\rho\,[v_0,\ldots,v_{n_0}]\,\Phi\,\sigma.\ \mathcal{E}[\![\mathtt{E}_0^{\mathtt{I}_0}]\!]([\mathtt{I}_{1_0} \mapsto v_1,\ldots,\mathtt{I}_{n_0} \mapsto v_{n_0}]\rho)\,\Phi\,\sigma]$$
$$(\ldots,\mathcal{L}[\![\mathtt{E}_m^{\mathtt{I}_m}]\!]([\mathtt{I}_m \mapsto \lambda\,\rho\,[v_0,\ldots,v_{n_m}]\,\Phi\,\sigma.$$
$$\mathcal{E}[\![\mathtt{E}_m^{\mathtt{I}_m}]\!]([\mathtt{I}_{1_m} \mapsto v_1,\ldots,\mathtt{I}_{n_m} \mapsto v_{n_m}]\rho)\,\Phi\,\sigma]\perp_{\mathit{ProcEnv}})\ldots))$$
$$\mathit{in}\ \mathit{let}\ \rho_0 = \mathit{fix}\ \lambda\,\rho.\ [\ldots,\mathtt{I}_i \mapsto \mathit{inClosure}(\mathtt{I}_i,\rho),\ldots]\lambda\,i.\ \perp_{\mathit{Val}}$$
$$\mathit{in}\ \mathit{let}\ (p,\rho) = (\rho_0\,\mathtt{I})\ \mathit{in}\ (\Phi_0\,p)\rho\,v^*\,\Phi_0\sigma_{\mathit{init}}$$

$$\mathcal{L} : \text{Pgm} \to \mathit{Proc\text{-}Env} \to \mathit{Proc\text{-}Env}$$
$$\mathcal{L}[\![\mathtt{C}]\!]\,\Phi = \Phi$$
$$\mathcal{L}[\![\mathtt{I}]\!]\,\Phi = \Phi$$
$$\mathcal{L}[\![\texttt{(if E}_0\ \texttt{E}_1\ \texttt{E}_2\texttt{)}]\!]\,\Phi = \mathcal{L}[\![\mathtt{E}_0]\!](\mathcal{L}[\![\mathtt{E}_1]\!](\mathcal{L}[\![\mathtt{E}_2]\!]\,\Phi))$$
$$\mathcal{L}[\![\texttt{(lambda (I}_1\ \ldots\ \texttt{I}_n\texttt{) E)}^p]\!]\,\Phi = \mathcal{L}[\![\mathtt{E}]\!]\,\Phi([p \mapsto \lambda\,\rho\,[v_1,\ldots,v_n]\,\Phi\,\sigma.$$
$$\mathcal{E}[\![\mathtt{E}]\!]([\mathtt{I}_1 \mapsto v_1,\ldots,\mathtt{I}_n \mapsto v_n]\rho)\,\Phi\,\sigma]\Phi)$$
$$\ldots$$

$$\mathcal{E} : \text{Expr} \to \mathit{Env} \to \mathit{Proc\text{-}Env} \to \mathit{Store} \to \mathit{Result}$$
$$\mathcal{E}[\![\mathtt{C}]\!]\,\rho\,\Phi\,\sigma = (\mathcal{C}[\![\mathtt{C}]\!],\sigma)$$
$$\mathcal{E}[\![\mathtt{I}]\!]\,\rho\,\Phi\,\sigma = (\rho\,\mathtt{I},\sigma)$$
$$\mathcal{E}[\![\texttt{(if E}_0\ \texttt{E}_1\ \texttt{E}_2\texttt{)}]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_0,\sigma_0) = \mathcal{E}[\![\mathtt{E}_0]\!]\,\rho\,\Phi\,\sigma$$
$$\mathit{in}\ \mathit{let}\ \mathit{GroundVal}(g) = v_0$$
$$\mathit{in}\ \mathit{let}\ \mathit{Bool}(b) = g$$
$$\mathit{in}\ (b \to (\mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi\,\sigma_0)\,[\!]\,(\mathcal{E}[\![\mathtt{E}_2]\!]\,\rho\,\Phi\,\sigma_0))$$
$$\mathcal{E}[\![\texttt{(lambda (I}_1\ \ldots\ \texttt{I}_n\texttt{) E)}^p]\!]\,\rho\,\Phi\,\sigma = (\mathit{inClosure}(p,\rho),\sigma)$$
$$\mathcal{E}[\![\texttt{(O E}_1\ \ldots\ \texttt{E}_n\texttt{)}]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_1,\sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi\,\sigma$$
$$\mathit{in}\ \ldots\ \mathit{let}\ (v_n,\sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\Phi\,\sigma_{n-1}$$
$$\mathit{in}\ \mathcal{O}[\![\mathtt{O}]\!][v_1,\ldots,v_n]\,\sigma_n$$
$$\mathcal{E}[\![\texttt{(E}_0\ \texttt{E}_1\ \ldots\ \texttt{E}_n\texttt{)}]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_0,\sigma_0) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi\,\sigma$$
$$\mathit{in}\ \ldots\ \mathit{let}\ (v_n,\sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\Phi\,\sigma_{n-1}$$
$$\mathit{in}\ \mathit{let}\ \mathit{Closure}(p,\rho_c) = v_0$$
$$\mathit{in}\ (\Phi\,p)\,\rho_c\,[v_1,\ldots,v_n]\,\Phi\,\sigma_n$$

Figure 4.8: Standard semantics of the higher-order language

The valuation functions of the standard semantics of the higher-order language is given in figure 4.8.

The semantics is quite usual for a Scheme-like language. The main exception is the globalized procedure environment, taking program points to procedure templates.

The valuation function $\mathcal{P}$ takes a program and generates a procedure environment and an environment. It then looks up the goal function and applies it to the initial arguments.

$\mathcal{L}$ generates a global procedure environment, assigning a procedure template to each procedure name and lambda-abstraction program-point. This environment is then propagated through $\mathcal{E}$ and procedure applications.

We have omitted $\mathcal{O}$, $\mathcal{C}$, and the equation for $\mathcal{E}$ on let-expressions, since they are essentially the same as the first-order version.

$\mathcal{E}$, the valuation function on expressions, is much as the first-order version. In the treatment of the conditional, we need an extra test, since the domain of values is now more complicated. The new rule for lambda-abstractions is very simple: since we have globalized the procedure templates, it just constructs the pair with the program point and the environment and injects it into the right summand. Correspondingly, the rule for application is relatively complex[2]. After determining the value of the operator, we have to look up the procedure template in the procedure environment and then apply it to the closure environment and the arguments.

4.2 A Grammar-Generating Analysis for First-Order Similix

In section 3.3 we outlined a grammar-generating analysis for Similix. In this section we formalize a first-order version of this analysis as an abstract interpretation. We first describe the domains corresponding to our proposed representation of residual programs, and then give the non-standard valuation functions of the analysis.

4.2.1 Abstract domains

Values

The elements of the domain of abstract values must be able to represent constructions in our concrete domain. We are interested in the constructions by the Similix syntax constructors (corresponding to the abstract syntax in figure 4.3). So for example in the case of the Similix constructor `sim-build-cond`, we want to be able to represent the result of applying this constructor as an "abstract residual conditional" with three abstract components. This implies that we need the entire domain of residual expressions in our abstract domain.

For other values such as natural numbers, we can be less precise. We can let an operation on numbers, such as +, return some representation of the set of all numbers instead of defining a precise addition on abstract values. This means that as an abstraction of the domain of natural numbers, we could use the two element domain. If we want to represent exact natural numbers that appear as syntactic constants in the source program, we can use the flat domain of numbers with a top element added to obtain a lattice.

If we treat the rest of the domains similarly, this gives us the following domain of abstract values:

$$Alternate_\tau = (Nat^\top + Bool^\top + Str^\top + Pair^\top + Res\text{-}Expr^\top)^\top$$

[2]No free lunch!

We will of course need to handle sets of such values. We represent this with lists, giving us:

$$Rhs_\gamma = Alternate_\gamma^*$$

We order the domain *Rhs* by subset inclusion on the sets of alternates, except that the top elements of the domains are taken to represent the entire corresponding set. So, for example, $[1, \mathbf{Nat}] \sqsubseteq_{Rhs} [\mathbf{Nat}]$, but $[\textit{"some-string"}, 1] \not\sqsubseteq [\mathbf{Nat}]$ (where Nat is $\top_{Nat}$).

Because a procedure call may return a non-terminal, non-terminals need to be values. So we have to add a flat domain of non-terminals to $Alternate_\gamma$:

$$Alternate_\gamma = (Nat^\top + Bool^\top + Str^\top + Pair^\top + Res\text{-}Expr^\top + Non\text{-}Terminal)$$

The addition of non-terminals introduce two new problems: residual expressions may contain non-terminals and the meaning of the non-terminals has to be specified.

Residual expressions may contain non-terminals because the non-terminal may represent a residual expression, and we do not want the abstract constructors to access the right hand side of the rule for the non-terminal. This means that we are actually considering a new domain of abstract residual expressions, $Res\text{-}Expr_\gamma$, where non-terminals can appear in place of any residual syntactic form.

Since the non-terminals represent the language that can be generated from them in a particular grammar, we have to add the grammar component to our abstract values, giving

$$Val_\gamma = Rhs_\gamma \times Grammar_\gamma$$

where a grammar is a list of rules and a rule is a non-terminal and a right hand side. For simplicity, we introduce special non-terminals to represent the top elements of the domains and assume that these are predefined to denote the language of all numbers, booleans, *etc.*

As with the program points, there is no particular need to organize the set of non-terminals as a domain, since it is ordered discretely and it is only used in constructions where a bottom element will appear anyway.

Note that the domain of values has no nice finiteness properties. We rely on the algorithm itself to terminate the analysis.

Even though we organize the domain of grammars as a list of rules, we consider it to be ordered by subset inclusion on the set of rules. This implies that merging by taking the union is also an upper bound on the languages generated (property 3).

So for two values, $(\alpha_1, \gamma_1) \sqsubseteq (\alpha_2, \gamma_2)$ if and only if $\alpha_1 \sqsubseteq \alpha_2 \wedge \gamma_1 \subseteq \gamma_2$.

Now, the constructors need to be defined on this domain of values. If we consider, for example, the primitive `_sim-build-primop-1`, then the abstraction of this primitive takes two abstract arguments, that are both a list of alternates and a grammar. It is sufficient that it operates on the alternates and just merges the two grammars, to make sure that all the non-terminals in the resulting construction are defined. Merging the grammars can safely be done by taking the union of the sets of rules.

So the constructor builds a list of alternates by building the applications of all the first alternates to all the second. So for example if the first list of alternates is [`add1`,`sub1`] and the second list of alternates is [`foo`,`bar`], the resulting list of alternates is [`(add1 foo)`, `(add1 bar)`, `(sub1 foo)`, `(sub1 bar)`]. Here foo and bar are supposed to be non-terminals representing the results of two procedures, while `add1` and `sub1` are terminals representing the strings `add1` and `sub1` in the target language and (and) are terminals representing (and) in the target language. Another approach would be to delay the construction (corresponding to introducing the { }-notation in the grammars,

so the example would give [({**add1** | **sub1**} {**foo** | **bar** })]). This would not make any theoretical difference, it is only a question of notation. It would, however, make the domains more complex, since we need to allow for the nested lists of alternates, so we will use the first approach so far.

After building the right hand side, the constructor merges the grammars and returns the abstract result.

Values
$v, (\alpha, \gamma) \in \mathit{Val}_\gamma = \mathit{Rhs} \times \mathit{Grammar}_\gamma$
$\gamma \in \mathit{Grammar}_\gamma = (\mathit{Non\text{-}Terminal} \times \mathit{Rhs})^*$
$\alpha, [a_1, \ldots, a_n] \in \mathit{Rhs} = \mathit{Alternate}^*$
$a \in \mathit{Alternate}_\gamma = (\mathit{Nat}^\top + \mathit{Bool}^\top + \mathit{Str}^\top + \mathit{Pair}_\gamma{}^\top + \mathit{Res\text{-}Expr}_\gamma + \mathit{Non\text{-}Term\text{-}Proc} + \mathit{Non\text{-}Term\text{-}Resid})^\top$
$\mathit{Pair}_\gamma = \mathit{Alternate}_\gamma \times \mathit{Alternate}_\gamma$
$n \in \mathit{Non\text{-}Term\text{-}Proc} = \mathit{Str}$
$n \in \mathit{Non\text{-}Term\text{-}Resid} = \mathit{Str}$
$\mathit{Non\text{-}Terminal} = \mathit{Non\text{-}Term\text{-}Proc} + \mathit{Non\text{-}Term\text{-}Resid}$

Environments
$\rho \in \mathit{Env}_\gamma = (\mathrm{Id} \to \mathit{Val}_\gamma)_\perp$

Stores
$\sigma \in \mathit{Store}_\gamma = \mathit{Val}_\gamma \times \mathit{Val}_\gamma$
$(v, \sigma) \in \mathit{Result}_\gamma = \mathit{Val}_\gamma \times \mathit{Store}_\gamma$

Procedures
$f \in \mathit{Proc}_\gamma = \mathit{Val}_\gamma^* \to \mathit{Store}_\gamma \to \mathit{Result}_\gamma$
$f \in \mathit{CProc} = (\{\lambda\, v^*\, \sigma.\, r \mid r \in \mathit{Result}_\gamma\} \cup \{\mathit{undef}_{\mathit{CProc}}\})_\perp$

Procedure environments
$\varphi \in \mathit{Penv}_\gamma = \mathit{Template}_\gamma \times \mathit{CEnv}$
$r \in \mathit{Template}_\gamma = (\text{Proc-Name} \to \mathit{CEnv} \to \mathit{Proc}_\gamma)_\perp$
$\varphi_c \in \mathit{CEnv} = \text{Proc-Name} \to \mathit{CProc}$

Figure 4.9: Grammar-generating semantics of the first-order language, domains

Procedures

To realize the idea of a memo-function, the grammar-generating semantics uses the domain of *constant* procedures, that is, procedures that given any argument return the same result, and the corresponding environment of constant procedure environments, that is, environments taking procedure names to constant procedures. The domain of constant procedures inherits the ordering on Result_γ. The constant procedure environment is the memo part of the procedure environment. It behaves as a "dynamically scoped" part of the procedure environment in the grammar-generating semantics.

The domain $\mathit{Template}_\gamma$ is used to represent the "statically scoped" part of the procedure environment. A template is a procedure environment with procedures that do not yet know which of the other procedures are constants. Given a procedure name and a constant procedure environment, however, it can return an actual procedure.

The domain $Penv_\gamma$ is defined as $Template_\gamma \times CEnv$ and ordered with the usual pairwise ordering. The grammar-generating semantics looks up a procedure name in the environment of constant procedures and only if it does not find it there will it look in the $Template_\gamma$ part. In the $Template_\gamma$ part the identifier is bound to a function from constant procedure environments to procedures. So in order to get a procedure, this function is applied to the current constant procedure environment. This way a dynamic updating of the constant part of a procedure environment will be propagated through the calls.

The domains of the grammar-generating semantics are given in figure 4.9.

4.2.2 Valuation functions

Figure 4.10 gives the grammar-generating semantics of the first-order language.

Some of it follows the usual pattern of abstract interpretation: the non-standard interpretation of the conditional joins the results of the two branches, except in the case where the abstract value happens to be either the terminal **True** or the terminal **False**, indicating that the test is really redundant.

The merging is done by considering the list of alternates in the right hand sides and the list of grammar rules in the grammars as sets and unioning them together to give new sets of alternates and rules.

The abstract procedure environment consists of a template part and a constant procedure environment part. The template part is constructed in $\mathcal{P}_\gamma$ as a fixed point. It binds each procedure name to a function that takes a constant procedure environment to a procedure. The procedure is constructed by the auxiliary ***mk-proc*** from the procedure name, the procedure template and the procedure environment.

When a procedure is applied in the valuation function $\mathcal{E}$, we first look up the name in the constant procedure environment. If a constant procedure is found there, we use this procedure. If the name is not in the constant procedure environment, we get a procedure by looking the name up in the template and applying the result to the constant procedure environment.

We then evaluate the arguments. Even though they are not actually used in the semantics of the procedure body, this is useful for the extension that we will design.

Finally we apply the procedure (either the constant one or the actual one) to the list of arguments and the store.

In the actual procedure, as defined in the auxiliary ***mk-proc***, we first generate a new non-terminal. We then generalize the arguments to the right hand side containing just $\top_{Alternate}$, which we call **Val**. We get a result by taking the fixed point over applying the procedure template to the procedure environment where the constant procedure environment is extended with the constant procedure that returns the result. This gives a result, and we add the rule with the non-terminal of the procedure and that result to the grammar.

The intention is that the denotation of a procedure body is determined in a procedure environment where the procedure is bound to a constant procedure. The constant procedure always returns the abstract result of applying the procedure, that is, the non-terminal representing the procedure and the grammar where that non-terminal is bound to the denotation of the procedure body.

Note that where the standard semantics has a fixed point over the denotation of the procedure (in the procedure environment), the non-standard semantics has a fixed point over the denotation of the ***body*** of the procedure.

Because the procedure is replaced by a constant procedure, there is no recursion. Furthermore, the fixed-point over the body of the procedure is trivial, since the arguments are generalized, so the

$$\mathcal{P}_\gamma : \text{Pgm} \to \text{Proc-Name} \to \mathit{Val}^\xi_\gamma \to \mathit{Val}_\gamma$$

$$\mathcal{P}_\gamma[\![((\texttt{define}\ (\mathrm{F}_1\ \mathrm{I}_{1_1}\ \ldots\ \mathrm{I}_{n_1})\ \mathrm{E}_1)\ \ldots\ (\texttt{define}\ (\mathrm{F}_m\ \mathrm{I}_{1_m}\ \ldots\ \mathrm{I}_{n_m})\ \mathrm{E}_m))]\!]\,\mathrm{F}\,v^* =$$
$$\mathit{let}\ \tau = \mathit{fix}\ \lambda\,\tau.\ [\mathrm{F}_1 \mapsto \mathit{mk\text{-}proc}\,\mathrm{F}_1\,(\lambda\,\varphi[v_1,\ldots,v_{n_1}].\ \mathcal{E}_\gamma[\![\mathrm{E}_1]\!][\mathrm{I}_{1_1} \mapsto v_1,\ldots,\mathrm{I}_{n_1} \mapsto v_{n_1}]\lambda\,i.\ \perp_{\mathit{Val}_\gamma}\varphi)\,\tau,\ \ldots,$$
$$\mathrm{F}_n \mapsto \mathit{mk\text{-}proc}\,\mathrm{F}_n\,(\lambda\,\varphi[v_1,\ldots,v_{n_m}].\ \mathcal{E}_\gamma[\![\mathrm{E}_m]\!][\mathrm{I}_{1_m} \mapsto v_1,\ldots,\mathrm{I}_{n_m} \mapsto v_{n_m}]\lambda\,i.\ \perp_{\mathit{Val}_\gamma}\varphi)\,\tau]\lambda\,f.\ \lambda\,\varphi.\ \perp_{\mathit{Proc}_\gamma}$$
$$\mathit{in}\ ((\tau\,\mathrm{F}(\lambda\,\mathrm{F}.\ \mathit{undef}_{\mathit{CProc}}))\,v^*\sigma_{\mathit{init}})\ \lfloor\ \rfloor$$

$$\mathcal{E}_\gamma : \text{Expr} \to \mathit{Env}_\gamma \to \mathit{Penv}_\gamma \to \mathit{Store}_\gamma \to \mathit{Result}_\gamma$$

$$\mathcal{E}_\gamma[\![\mathrm{C}]\!]\,\rho\,\varphi\,\sigma = (\mathcal{C}_\gamma[\![\mathrm{C}]\!],\sigma)$$

$$\mathcal{E}_\gamma[\![\mathrm{I}]\!]\,\rho\,\varphi\,\sigma = (\rho\,\mathrm{I},\sigma)$$

$$\mathcal{E}_\gamma[\![(\texttt{if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2)]\!]\,\rho\,\varphi\,\sigma = \mathit{let}\ ((\alpha,\gamma),\sigma_0) = (\mathcal{E}_\gamma[\![\mathrm{E}_0]\!]\,\rho\,\varphi\,\sigma)$$
$$\mathit{in}\ (\alpha = [\mathbf{True}]) \to (\mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma_0)$$
$$\lfloor\ (\alpha = [\mathbf{False}]) \to (\mathcal{E}_\gamma[\![\mathrm{E}_2]\!]\,\rho\,\varphi\,\sigma_0)$$
$$\lfloor\ \mathit{join}\,(\mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma_0)\,(\mathcal{E}_\gamma[\![\mathrm{E}_2]\!]\,\rho\,\varphi\,\sigma_0)$$

$$\mathcal{E}_\gamma[\![(\mathrm{O}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\,\rho\,\varphi\,\sigma = \mathit{let}\ (v_1,\sigma_1) = (\mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma)$$
$$\mathit{in}\ \ldots\ \mathit{let}\ (v_n,\sigma_n) = (\mathcal{E}_\gamma[\![\mathrm{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1})$$
$$\mathit{in}\ \mathcal{O}_\gamma[\![\mathrm{O}]\!]\,[v_1,\ldots,v_n]\,\sigma_n$$

$$\mathcal{E}_\gamma[\![(\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\,\rho(\tau,\varphi_c)\,\sigma = \mathit{let}\ f = (((\varphi_c\,\mathrm{F}) = \mathit{undef}_{\mathit{CProc}}) \to (\tau\,F\,\varphi_c)\ \lfloor\ \varphi_c\,\mathrm{F})$$
$$\mathit{in}\ \mathit{let}\ (v_1,\sigma_1) = (\mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma)$$
$$\mathit{in}\ \ldots \mathit{let}\ (v_n,\sigma_n) = (\mathcal{E}_\gamma[\![\mathrm{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1})\ \mathit{in}\ f\,[v_1,\ldots,v_n]\,\sigma_n$$

$$\mathcal{E}_\gamma[\![(\texttt{let}\ ((\mathrm{I}\ \mathrm{E}_1))\ \mathrm{E}_2)]\!]\,\rho\,\varphi\,\sigma = \mathit{let}\ (v,\sigma') = \mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma\ \mathit{in}\ \mathcal{E}_\gamma[\![\mathrm{E}_2]\!]\,[\mathrm{I} \mapsto v]\rho\,\varphi\,\sigma'$$

$$\mathcal{O}_\gamma : \text{Op} \to \mathit{Val}^\xi_\gamma \to \mathit{Store}_\gamma \to \mathit{Result}_\gamma$$

$$\mathcal{O}_\gamma[\![\texttt{cons}]\!]\,[(\alpha,\gamma)\,(\alpha',\gamma')]\,\sigma = (([\mathbf{Val}],\gamma_{\mathit{empty}}),\sigma)$$

$$\mathcal{O}_\gamma[\![\texttt{car}]\!]\,[([a_1,\ldots,a_n],\gamma)]\,\sigma = (([\mathbf{Val}],\gamma_{\mathit{empty}}),\sigma)$$

$$\ldots$$

$$\mathcal{O}_\gamma[\![\texttt{\_sim-build-primop1}]\!]\,[([\ldots,a_i,\ldots],\gamma),([\ldots,a'_j,\ldots],\gamma')]\,\sigma =$$
$$(([\ldots,\mathit{inRes\text{-}Expr}(\mathit{primop1},a_i,a'_j),\ldots],\gamma\cup\gamma'),\sigma)$$

$$\ldots$$

$$\mathcal{O}_\gamma[\![\texttt{\_sim-generate-proc-name!}]\!]\,[(\alpha,\gamma),(\alpha',\gamma')]\,\sigma =$$
$$\mathit{let}\ a = \mathit{gen\text{-}new\text{-}name}_\gamma\,(\alpha,\gamma)$$
$$\mathit{in}\ ((([\mathit{inPair}(\mathit{inPair}(a,\mathbf{Nat}),[\mathbf{Bool}])]),\gamma\cup\gamma'),\sigma)$$

where

$$\mathit{mk\text{-}proc} : \text{Proc-Name} \to (\mathit{Penv}_\gamma \to \mathit{Proc}_\gamma) \to \mathit{Template}_\gamma \to \mathit{CEnv} \to \mathit{Proc}_\gamma$$

$$\mathit{mk\text{-}proc} = \lambda\,\mathrm{F}\,\pi\,\tau.\ \lambda\,\varphi_c.\ \lambda\,v^*\sigma.\ \mathit{let}\ \alpha' = \mathit{mk\text{-}non\text{-}terminal}\ \mathrm{F}\ \sigma$$
$$\mathit{in}\ \mathit{let}\ r_\gamma = \mathit{fix}\ \lambda\,r_\gamma.\ \pi\ (\tau,([\mathrm{F} \mapsto \lambda\,v^*\sigma.\ (\mathit{abstract\text{-}res}\,\alpha'\,r_\gamma)]\varphi_c))$$
$$(\mathit{generalize\text{-}args}\,v^*)\,(\mathit{generalize\text{-}store}\,\sigma)$$
$$\mathit{in}\ (\mathit{abstract\text{-}res}\,\alpha'\,r_\gamma)$$

$$\mathit{abstract\text{-}res} = \lambda\,[n_0,n_1,n_2].\ \lambda\,((\alpha_0,\gamma_0),((\alpha_1,\gamma_1),(\alpha_2,\gamma_2))).$$
$$((n_0,\{(n_0,\alpha_0)\}\cup\gamma_0),((n_1,\{(n_1,\alpha_1)\}\cup\gamma_1),(n_2,\{(n_2,\alpha_2)\}\cup\gamma_2)))$$

$$\mathit{join} = \lambda\,(\alpha,\gamma)\,(\alpha',\gamma').\ (\alpha\cup\alpha',\gamma\cup\gamma')$$

$$\mathit{generalize\text{-}arg} = \lambda\,(\alpha,\gamma).\ (\mathbf{Val},\gamma_{\mathit{empty}})$$

$$\mathit{generalize\text{-}store} = \lambda\,\sigma.\ ((\mathbf{Val},\gamma_{\mathit{empty}}),(\mathbf{Val},\gamma_{\mathit{empty}}))$$

Figure 4.10: Grammar-generating semantics of the first-order language

value of the procedure is the same at each iteration.

Most primitive operations are abstracted so that in most cases they simply give the generic abstract value **Val**, containing no information except that it is a value. So the abstraction of *cons* does not produce a pair, *etc.* Pairs can occur, however, for example as constants in the text of the program, in which case the abstraction of *car* might take the car of the pair. The notable exceptions to this simplistic view are the constructors on syntax trees and the primitive adding residual procedures to the store.

Other, more precise, interpretations of the primitives are of course also possible.

The abstractions of the syntax-constructors construct "abstract syntax" in the right hand side, as mentioned above.

The abstractions of the book-keeping primitives are intended to do a safe approximation of the book-keeping. So for example the semantics of _sim-generate-proc-name! always returns the flag **Bool**, independently of the values in the seenbefore list.

The memo-function scheme that we use here is rather simple-minded, since the memo-function is never returned. When a procedure is called, the memo-function is extended and propagated downwards in the treatment of the call, which ensures termination. But when the procedure has been analyzed, the extension is forgotten. This is of course rather inefficient. It can be remedied, for example, by globalizing the memo-function.

We have chosen this design to simplify the safety proof for the analysis. One we have a proof of safety for this version, it should be relatively simple to prove safety of a globalized version with respect to this version, based on extensionality.

4.2.3 Generating non-terminals for procedure parameters

The grammar-generating semantics described above ensures termination by generalizing the arguments to procedure calls to the top element in the domain of values, so that each procedure will be analyzed at most once. This of course causes a loss of information in the analysis that can be quite significant.

This loss can be modified by allowing a larger, but still finite, set of actuals. For example a string could be generalized to the top element of the domain *Str*, retaining at least some of the information.

A more general approach, however, is to generate non-terminals and rules in the grammar corresponding to each parameter of each procedure. Then the procedure would still be analyzed only once, since a reference to a parameter would return a non-terminal, which would be the same, no matter what the actual argument is. So any call to the procedure after the first would only result in adding rules to the grammar, corresponding to the new (abstract) values of the parameters. In particular any parameter that was increased in a recursive call (such as an accumulator) would result in a recursive rule.

Example:

The program

```
(define (f a b)
   (if (zero? a)
       b
       (f (sub1 a) (cons a b))))
```

would correspond to the grammmar

```
PROC-f --> PAR-b | PROC-f
PAR-a --> Val
PAR-b --> (PAR-a . PAR-b) |  Val
```

If we analyze a program where f is used in a context such as

```
(define (g x)
    (f x ()))
```

we would get the grammar

```
PROC-g --> PROC-f
PROC-f --> PAR-b | PROC-f
PAR-a --> Val | PAR-x
PAR-b --> (PAR-a . PAR-b) |  ()
```

and we could say that the result of `f` is a list and its elements correspond to the parameter `a`.

Fortunately, it is sufficient to make some rather limited modifications to the grammar-generating semantics to obtain this effect.

The constant environment now has to remember which arguments the procedure has been called with as well as its name. If the procedure is already defined in the constant environment, but with a different set of arguments, we must add the current arguments to the rules corresponding to the procedure parameters. If the procedure is not in the constant environment, we proceed as before.

Rules for the parameters are added at application time. When a procedure is applied to a right hand side r and a grammar γ, we extend γ with a rule for the non-terminal corresponding to the formal parameter with right hand side r and bind the formal parameter to the value with the non-terminal as right hand side and the extended grammar.

Grammar-generating semantics with non-terminals for parameters

$$a \in \mathit{Alternate}^{nt}_{\gamma} = (\mathit{Nat}^{\top} + \mathit{Bool}^{\top} + \mathit{Str}^{\top} + \mathit{Pair}^{\top}_{\gamma} + \mathit{Res\text{-}Expr}_{\gamma} + \mathit{Non\text{-}Term\text{-}Proc} + \mathit{Non\text{-}Term\text{-}Resid} + \mathit{Non\text{-}Term\text{-}Param})^{\top}$$
$$n \in \mathit{Non\text{-}Term\text{-}Param} = \mathit{Str}$$
$$\varphi_c \in \mathit{CEnv}^{nt} = \text{Proc-Name} \times \mathit{Val}^{*}_{\gamma} \times \mathit{Store}_{\gamma} \rightarrow (\mathit{Result}_{\gamma} + \mathit{Undef})_{\perp}$$
$$\mathit{Undef} = \mathit{Unit}$$

Figure 4.11: Grammar-generating semantics with non-terminals for parameters, modified domains

The new domains in the version generating non-terminals for the parameters is given in figure 4.11. The remaining domains are unchanged.

Because the constant environment is now supposed to remember the arguments that a procedure has been called with, and only return the saved constant function in the case where the arguments have been seen before, the domain has to be modified accordingly. We can now no longer just look up the name of the procedure and return a constant function or a symbol that it is not found. Instead the domain must look much like for normal procedure environments, taking procedure names and arguments to a result. So now we look up a procedure and its arguments. If it is found, a (constant) result is returned. If it is not found, the element of *Unit* is returned.

$$\mathcal{E}^{nt}_{\gamma}[\![(\mathrm{F}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n)]\!]\,\rho\,(\tau,\varphi_c)\,\sigma = \mathit{let}\ (v_1,\sigma_1) = (\mathcal{E}^{nt}_{\gamma}[\![\mathrm{E}_1]\!]\,\rho\,\varphi\,\sigma)$$
$$\mathit{in}\ \ldots\ \mathit{let}\ (v_n,\sigma_n) = (\mathcal{E}^{nt}_{\gamma}[\![\mathrm{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1})$$
$$\mathit{in}\ \mathit{let}\ r = (\varphi_c\,\mathrm{F}\,[v_1,\ldots,v_n]\,\sigma_n)$$
$$\mathit{in}\ \mathit{cases}\ r\ \mathit{of}$$
$$\mathit{isUndef}() \rightarrow (\tau\,\mathrm{F}\,\varphi_c)\,[v_1,\ldots,v_n]\,\sigma_n$$
$$\| \ \mathit{isResult}(r) \rightarrow r$$

where

$$\mathit{mk\text{-}proc} = \lambda\,\mathrm{F}\,\pi\,\tau.$$
$$\lambda\,\varphi_c.\,\lambda\,v^*\,\sigma.\ \mathit{let}\ \alpha' = \mathit{mk\text{-}non\text{-}terminal}\ \mathrm{F}\ \sigma$$
$$\mathit{in}\ \mathit{let}\ w^* = \mathit{abstract\text{-}arguments}\ \alpha'\ v^*$$
$$\mathit{in}\ \mathit{let}\ \sigma' = \mathit{abstract\text{-}store}\ \alpha'\ \sigma$$
$$\mathit{in}\ \mathit{let}\ r_\gamma = \mathit{fix}\,\lambda\,r_\gamma.\ \pi\,(\tau,([[\mathrm{F},v^*,\sigma]\mapsto(\mathit{abstract\text{-}res}\ \alpha'\ r_\gamma)]\varphi_c))\ w^*\ \sigma'$$
$$\mathit{in}\ (\mathit{abstract\text{-}res}\ \alpha'\ r_\gamma)$$

$$\mathit{abstract\text{-}argument} = \lambda\,[n_0,n_1,n_2]\,(\alpha_0,\gamma_0)\,i.\ (n_0 :: i, \{(n_0 :: i,\alpha_0)\} \cup \gamma_0)$$

Figure 4.12: Grammar-generating semantics with non-terminals for the parameters

Figure 4.12 gives the lines in the semantics that have to be changed to generate non-terminals for the parameters.

Note that with this extension, we have lost the property that the fixed-point is trivial. Now changes in the arguments may effect changes in the result. The changes will not be in the rule for the procedure, though, since any use of a parameter in the body of the procedure will appear as the non-terminal of that parameter[3]. Instead, changes in the arguments will appear as changes in the rule for the parameters. As long as we maintain a syntactic limit on the possible right-hand sides for any given program, this will not affect termination. This is discussed in more detail in section 6.2.

Since the operation ***abstract-argument*** is safe with respect to the language generated by the value, this extension of the grammar-generating semantics will not affect the proof of safety.

4.3 Extending the Analysis to Higher-Order Programs

4.3.1 The abstract interpretation

We extend the grammar-generating analysis of the previous section to handle higher-order programs by combining it with a closure analysis.

In order to obtain a first-order representation of the higher-order values we use a closure representation, that is, the denotation of a lambda abstraction is a pair containing the actual denotation of the abstraction (a procedure template) and an environment binding the free variables of the abstraction.

As in the first-order analysis, we want to generate non-terminals for the procedures. We assume that abstract closures are essentially the same kind of objects as abstract procedures. So we also want to generate non-terminals (and rules) for each abstract closure in the program. This means that when the analysis encounters an application of an abstract closure, it will symbolically apply

[3]This only holds because the non-standard semantics of the primitive operations is so simple. If we allowed a more precise interpretation of the predicates and the destructors, that could access the right-hand side of a rule, then changes in the parameters could also cause changes in the rule for the procedure

it, using the same schema as in the first-order case. How expensive this becomes, in terms of complexity of the analysis, can be limited by limiting the "granularity" of the closure environments. We assume (without loss of generality) that all the procedure names and formal parameters in the program are distinct. Then we can use these and the program points as non-terminals.

In order to make this work, we need to integrate procedures into our domain of values, while still maintaining our scheme for generating grammar-rules. This imposes some restrictions on our design

- we need to have a safe equality predicate on procedures, in order to be able to determine if we have already generated a rule for a particular procedure.
- since environments are used in finding the denotation of lambda-abstractions, we need a finite representation of environments matching our choice of representation for abstract closures.

In fulfilling these restrictions, we can make some (more or less) arbitrary choices:

- which abstract closures do we consider to be identical, *i.e.*, we want to generate only one rule for all of them.
- which level of detail do we want in the abstract environments.
- how do we view the free variables of a lambda-abstraction.

Free variables

To start at the bottom, we have two options on what to do with the free variables of the lambda abstractions: we can either consider that each distinct closure (with different environments binding the free variables) defines a "new" procedure, worthy of its own non-terminal, or we can generate only one non-terminal for each syntactic lambda and add to the production every time a new environment is encountered[4]. We have chosen the latter, since it will probably give slightly less verbose output, but this can easily be changed if it appears to obstruct precision too much.

Environment representation

Since we are assuming that the grammar representation and the generation of non-terminals assure that only finitely many ground values will be used during the analysis of any given program, the problem of finding a finite representation of environments lies with the closures. Infinitely increasing environments based on finitely many ground values can be generated for example by a looping program with a continuation.

There are several solutions to this problem, see for example Shivers [Shi91]. The basic idea is that a closure does not contain an environment, only a pointer to an environment. Thus the potentially infinite nesting of closures and environments can be replaced by circular pointers in the abstraction.

Distinguishing between call-sites

When generating non-terminals for procedures and lambda-abstractions, we have another, similar, choice: we can either generate only one non-terminal for each definition or we can generate a new

[4] Instead of adding to the production, we could also generate "parameter" non-terminals for the free variables in the lambda abstraction. This would effectively correspond to "lambda-lifting on the fly".

non-terminal for each syntactic occurrence of a call to the procedure/lambda-abstraction. This could be done by using program points to identify the call-sites.

Clearly the latter can be expected to lead to more verbose output, since it generates more non-terminals. It does, however, avoid the merging of information coming from two different call-sites, giving a kind of "tensor-product effect"[NN92]. Instead of taking all possible combinations of the possible values of parameters, the values of parameters are kept with the values from the same call-site.

Since the former is the simpler, and an extension to distinguish between call-sites does not seem to make any theoretical difference, we use the former version.

4.3.2 Domains

Abstract representation of closures

To add the closures to our domain of abstract values, we first observe that since the language is untyped, we can have the same variable bound to both ground values and closures at different steps in the evaluation. This means we need an abstract representation of sets of values containing both closures and ground values. This is obtained by letting abstract values be a *product* of abstract closures and abstract ground values:

$$Val_\gamma = Closure_\gamma^* \times Rhs_\gamma \times Grammar_\gamma$$

Such a representation is sufficient because of the restrictions we have put on the use of closures. Because we have trivialized most primitive destructors, allowing constructions on closures would result in total loss of information about the closure (the analysis cannot interpret the destructors that are used to access the structure). This is not a problem with the current version of Similix, but the problem might occur in later versions. In that case, we would have to modify the design.

We use the correspondence between program points and procedure templates to globalize the procedure templates. Instead of a closure consisting of a procedure template and an environment, it consists of a program point and an environment. At application time, the program point is used to look up the procedure template in a global procedure environment. Assuming the unique correspondence between program points and procedure templates, this globalization step is quite easy to prove correct for any given program. Note that if we assume that the names of all global procedures are distinct (which Similix requires), then we have a similar representation for the global procedures.

This gives us a computable equality predicate on closures if we have a finite representation of environments.

Environments

We design a relatively detailed representation of environments where a closure contains an environment, but the closures in *that* environment have only a pointer to an environment. So in environment extension, when we bind an identifier to a closure, we only bind it to the program point of the closure, not the environment. This gives us a high level of detail, but ensures that the environments are finite, since there are no nested closures. So our simplified environments look like this:

$$BaseEnv_\gamma = (\mathrm{Id} \times BaseVal_\gamma)^*$$

$$BaseVal_\gamma = Program\text{-}Point^* \times Rhs_\gamma \times Grammar_\gamma$$

But when we look up an identifier, we expect to get a value, *i.e.*, a closure, not just a program point. So which environment can we give to the program point after we have looked it up to make it a closure?

To supply this, we add a component to the environment, that for each program point saves an upper bound of the environments corresponding to this program point:

$$EnvEnv_\gamma = (Program\text{-}Point \times BaseEnv_\gamma)^\bullet$$

So an environment contains a base environment and an environment of base environments:

$$Env_\gamma = BaseEnv_\gamma \times EnvEnv_\gamma$$

Note that a more common solution is to globalize the structure that assigns an (upper bound) environment to each program point (the domain $EnvEnv_\gamma$) (what Shivers [Shi91] calls a 0CFA analysis). This potentially leads to less precision, but it is not clear whether the loss is significant. We have chosen a localized approach here in keeping with our general strategy of keeping information local whenever possible.

The abstract domains of the analysis are given in figure 4.13.

$$\begin{array}{l}
v_\gamma \in Val_\gamma = Closure_\gamma^\bullet \times Rhs_\gamma \times Grammar_\gamma \\
c \in Closure_\gamma = Program\text{-}Point \times Env_\gamma \\
b \in BaseVal_\gamma = Program\text{-}Point^\bullet \times Rhs_\gamma \times Grammar_\gamma \\
r \in Rhs_\gamma = Alternate_\gamma^\bullet \\
a \in Alternate_\gamma = (Nat^\top + Bool^\top + Str^\top + Pair_\gamma^\top + Res\text{-}Expr^\top + Non\text{-}Terminal)^\top \\
(a_1, a_2) \in Pair_\gamma = Alternate_\gamma \times Alternate_\gamma \\
\gamma \in Grammar_\gamma = (Non\text{-}Terminal \times Rhs_\gamma)^\bullet \\
\rho_\gamma \in Env_\gamma = BaseEnv_\gamma \times EnvEnv_\gamma \\
\beta \in BaseEnv_\gamma = (\mathrm{Id} \times BaseVal_\gamma)^\bullet \\
\kappa \in EnvEnv_\gamma = (Program\text{-}Point \times BaseEnv_\gamma)^\bullet \\
\Phi, (\mu, \varphi) \in Proc\text{-}Env_\gamma = Memo \times (Program\text{-}Point \rightharpoonup Proc\text{-}Template_\gamma) \\
\mu \in Memo = ((Program\text{-}Point \times Env_\gamma \times Val_\gamma^\zeta \times Store_\gamma) \times (Val_\gamma \times Store_\gamma))^\bullet \\
\pi \in Proc\text{-}Template_\gamma = Env_\gamma \rightharpoonup Proc_\gamma \\
f \in Proc_\gamma = Val_\gamma^\zeta \rightharpoonup Proc\text{-}Env_\gamma \rightharpoonup Store_\gamma \rightharpoonup Result_\gamma \\
\sigma \in Store_\gamma = Val_\gamma \times Val_\gamma \\
(v, \sigma) \in Result_\gamma = Val_\gamma \times Store_\gamma
\end{array}$$

Figure 4.13: Grammar-generating semantics of the higher-order language, domains

4.3.3 Valuation functions

The grammar-generating semantics for the higher-order language is given in figure 4.14.

As in the standard semantics, the valuation function $\mathcal{P}_\gamma$ takes a program and generates an environment and a procedure environment. It then looks up the goal function and applies it to the initial arguments.

The valuation function $\mathcal{E}_\gamma$ on expressions does the usual things for constants and identifiers, and the conditional is the usual abstract interpretation joining of the two branches.

Similarly the auxiliary for application is the join of applying all the possible closures. Since applying a ground value would be an error not producing any valid result, we do not need to consider this case.

As in the first-order analysis, the treatment of application is the key part of the analysis. It is here the introduction of non-terminals take place and the only place anything is added to the grammar part of an abstract value (using the auxiliary *abstract-res*). It is also here the memo-function is extended. As in the first-order analysis, we take the fixed point of the computation of the result in order to be able to extend the memo-function before we have actually computed the value we want to extend it with. This is possible because the fixed point operator here is in fact only a cosmetic device: it ensures the well-definedness of the non-terminals, but since we do not access the grammar part of a value, it does not correspond to an actual (non-trivial) fixed point iteration.

Constructing closures and constructing the procedure environment is straightforward.

The environment handling basically implement the "two-component" environment scheme that we described above.

As before, we leave the valuation function $\mathcal{O}_\gamma$, for primitive operations, unspecified.

The semantics uses several auxiliary functions, some of which are given in figure 4.15.

Of particular interest are *extend-env*$_\gamma$ and *lookup-env*$_\gamma$, that realize the environment design outlined above. Application of procedures has become rather voluminous, so we give it as a separate function. It implements essentially the same rule-generating scheme as before, but now that we potentially have a set of procedures to apply, we have to merge their results.

The auxiliary combinator *seenb4?* is similar to the one used by Similix: it looks up the procedure and its arguments in the memo-function μ and returns the corresponding entry if it finds them. Note that it would also be safe to let *seenb4?* return a saved entry if the argument was smaller than something already stored in the memo-function.

$$\mathcal{P}_\gamma : \mathrm{Pgm} \to \mathrm{Id} \to Val\xi_\gamma \to Val_\gamma$$

$$\mathcal{P}_\gamma[\![\texttt{(define (}\mathrm{I}_0\ \mathrm{I}_{1_0}\ \ldots\ \mathrm{I}_{n_0}\texttt{)}\ \mathrm{E}_0^{\mathrm{I}_0}\texttt{)}\ \ldots\ \texttt{(define (}\mathrm{I}_m\ \mathrm{I}_{1_m}\ \ldots\ \mathrm{I}_{n_m}\texttt{)}\ \mathrm{E}_0^{\mathrm{I}_m}\texttt{)}]\!]\, \mathrm{I}\, v^* =$$
$$\quad \mathit{let}\ \varphi_0 = \mathcal{L}_\gamma[\![\mathrm{E}_0^{\mathrm{I}_0}]\!]([\mathrm{I}_0 \mapsto \lambda\, \rho\, [v_1, \ldots, v_{n_0}]\, \Phi\, \sigma.\ \mathcal{E}_\gamma[\![\mathrm{E}_0^{\mathrm{I}_0}]\!]\,(\mathit{extend\text{-}env}_\gamma\ [\mathrm{I}_{1_0}, \ldots, \mathrm{I}_{n_0}]\,[v_1, \ldots, v_{n_0}]\,\rho)\,\Phi\,\sigma]$$
$$\qquad (\ldots \mathcal{L}_\gamma[\![\mathrm{E}_m^{\mathrm{I}_m}]\!]([\mathrm{I}_m \mapsto \lambda\, \rho\, [v_{1_m}, \ldots, v_{n_m}]\, \Phi\, \sigma.$$
$$\qquad\qquad \mathcal{E}_\gamma[\![\mathrm{E}_m^{\mathrm{I}_m}]\!]\,(\mathit{extend\text{-}env}_\gamma\ [\mathrm{I}_{1_m}, \ldots, \mathrm{I}_{n_m}]\,[v_1, \ldots, v_{n_m}]\,\rho)\,\Phi\,\sigma]$$
$$\qquad\qquad \bot_{ProcEnv}) \ldots))$$
$$\quad \mathit{in}\ \mathit{let}\ \rho_0 = \mathit{let}\ \beta = [\ldots, (p_i, ([p_i], [\,], \gamma_{empty})), \ldots]$$
$$\qquad\qquad \mathit{in}\ (\beta, [\ldots, (p_i, \beta), \ldots])$$
$$\qquad \mathit{in}\ (\mathit{apply}_\gamma\ (\mathit{lookup\text{-}env}_\gamma\ \mathrm{I}\ \rho_0)\, v^*\, (\mu_{init}, \varphi_0)\, \sigma_{init})\ \lfloor\, 1$$

$$\mathcal{L}_\gamma : \mathrm{Pgm} \to (\mathit{Program\text{-}Point} \to \mathit{Proc\text{-}Template}_\gamma) \to (\mathit{Program\text{-}Point} \to \mathit{Proc\text{-}Template}_\gamma)$$
$$\mathcal{L}_\gamma[\![\mathrm{C}]\!]\,\varphi = \varphi$$
$$\mathcal{L}_\gamma[\![\mathrm{I}]\!]\,\varphi = \varphi$$
$$\mathcal{L}_\gamma[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!]\,\varphi = \mathcal{L}_\gamma[\![\mathrm{E}_0]\!](\mathcal{L}_\gamma[\![\mathrm{E}_1]\!](\mathcal{L}_\gamma[\![\mathrm{E}_2]\!]\,\varphi))$$
$$\mathcal{L}_\gamma[\![\texttt{(lambda (}\mathrm{I}_1\ \ldots\ \mathrm{I}_n\texttt{)}\ \mathrm{E}\texttt{)}^p]\!]\,\varphi =$$
$$\mathcal{L}_\gamma[\![\mathrm{E}]\!]\,\varphi([p \mapsto \lambda\, \rho\, [v_1, \ldots, v_n]\, \Phi\, \sigma.\ \mathcal{E}_\gamma[\![\mathrm{E}]\!]([\mathrm{I}_1 \mapsto v_1, \ldots, \mathrm{I}_n \mapsto v_n]\rho)\,\Phi\,\sigma]\varphi)$$
$$\ldots$$
$$\mathcal{E}_\gamma : \mathrm{Expr} \to \mathit{Env}_\gamma \to \mathit{Proc\text{-}Env}_\gamma \to \mathit{Store}_\gamma \to \mathit{Result}_\gamma$$
$$\mathcal{E}_\gamma[\![\mathrm{C}]\!]\,\rho\,(\mu, \varphi)\,\sigma = (\mathcal{C}_\gamma[\![\mathrm{C}]\!], \sigma)$$
$$\mathcal{E}_\gamma[\![\mathrm{I}]\!]\,\rho\,(\mu, \varphi)\,\sigma = (\mathit{lookup\text{-}env}_\gamma\ \mathrm{I}\ \rho, \sigma)$$
$$\mathcal{E}_\gamma[\![\texttt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\texttt{)}]\!]\,\rho\,(\mu, \varphi)\,\sigma = \mathit{let}\ (v_0, \sigma_0) = \mathcal{E}_\gamma[\![\mathrm{E}_0]\!]\,\rho\,(\mu, \varphi)\,\sigma$$
$$\qquad\qquad \mathit{in}\ \mathit{join\text{-}result}\,(\mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,(\mu, \varphi)\,\sigma_0)\,(\mathcal{E}_\gamma[\![\mathrm{E}_2]\!]\,\rho\,(\mu, \varphi)\,\sigma_0)$$
$$\mathcal{E}_\gamma[\![\texttt{(lambda (}\mathrm{I}_1\ \ldots\ \mathrm{I}_n\texttt{)}\ \mathrm{E}\texttt{)}^p]\!]\,\rho\,(\mu, \varphi)\,\sigma = (([(p, \rho)], [\,], \gamma_{empty}), \sigma)$$
$$\mathcal{E}_\gamma[\![\texttt{(O}\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n\texttt{)}]\!]\,\rho\,(\mu, \varphi)\,\sigma = \mathit{let}\ (v_1, \sigma_1) = \mathcal{E}_\gamma[\![\mathrm{E}_1]\!]\,\rho\,(\mu, \varphi)\,\sigma$$
$$\qquad\qquad \mathit{in}\ \ldots\ \mathit{let}\ (v_n, \sigma_n) = \mathcal{E}_\gamma[\![\mathrm{E}_n]\!]\,\rho\,(\mu, \varphi)\,\sigma_{n-1}$$
$$\qquad\qquad\qquad \mathit{in}\ \mathcal{O}_\gamma[\![\mathrm{O}]\!][v_1, \ldots, v_n]\,\sigma_n$$
$$\mathcal{E}_\gamma[\![\texttt{(}\mathrm{E}_0\ \mathrm{E}_1\ \ldots\ \mathrm{E}_n\texttt{)}]\!]\,\rho\,(\mu, \varphi)\,\sigma = \mathit{let}\ (v_0, \sigma_0) = \mathcal{E}_\gamma[\![\mathrm{E}_0]\!]\,\rho\,(\mu, \varphi)\,\sigma$$
$$\qquad\qquad \mathit{in}\ \ldots\ \mathit{let}\ (v_n, \sigma_n) = \mathcal{E}_\gamma[\![\mathrm{E}_n]\!]\,\rho\,(\mu, \varphi)\,\sigma_{n-1}$$
$$\qquad\qquad\qquad \mathit{in}\ \mathit{apply}_\gamma\ v_0[v_1, \ldots, v_n]\,(\mu, \varphi)\,\sigma_n$$

Figure 4.14: Grammar-generating semantics of the higher-order language

```
extend-env_γ : Id* → Val_ξ → Env_γ → Env_γ
extend-env_γ [..., I_i, ...] [..., ([..., (p_ij, (β_ij, κ_ij)), ...], r_i, γ_i), ...] (β, κ) =
  let κ' = join-κ [..., κ_ij, ..., κ]
  in let κ'' = extend-env-env [..., (p_ij, β_ij), ...] κ'
     in ([..., I_i ↦ ([..., p_ij, ...], r_i, γ_i), ...]β, κ'')

lookup-env_γ : Id → Env_γ → Val_γ
lookup-env_γ I (β, κ) = let ([..., p_i, ...], r, γ) = lookup I β
                        in ([..., (p_i, ((lookup p_i κ), κ)), ...], r, γ)

lookup i [..., (i, x_j), ...] = x_j

apply_γ : Val_γ → Val_ξ → Proc-Env_γ → Store_γ → Result_γ
apply_γ ([..., (p_i, ρ_i), ...], r, γ) v* (μ, φ) σ =
(join-results
[..., cases (seenb4? (p_i, ρ_i, v*, σ) μ) of
        isFound(v, σ) → (v, σ)
      [] isNotFound() →
          let w* = abstract-arguments p_i v*
          in let s' = abstract-store p_i σ
             in let (v_r, σ_r) =
                    fix λ (v_r, σ_r).
                        (φ p_i) ρ_i w* ([(p_i, ρ_i, v*, σ) ↦ abstract-res p_i (v_r, σ_r)]μ, φ)s'
                in (abstract-res p_i (v_r, σ_r)),
      ...])

seenb4? : Program-Point × Env × Val* × Store → Memo → (Found + NotFound)_⊥

join-result : Result_γ → Result_γ → Result_γ
join-result (v_1, σ_1) (v_2, σ_2) = (join-v v_1 v_2, join-σ σ_1 σ_2)

join-v : Val_γ → Val_γ → Val_γ
join-v (c_1*, r_1, γ_1) (c_2*, r_2, γ_2) = (c_1* ∪ c_2*, r_1 ∪ r_2, γ_1 ∪ γ_2)
```

Figure 4.15: Grammar-generating semantics of the higher-order language, auxiliary definitions

Chapter 5

Examples

In this chapter we give examples of the analyses applied to different programs. We first give a very small example, to illustrate the format. Then we give two examples of the first-order analysis applied to the generating extensions of Similix. Finally we give an example of the higher-order analysis. Since we have not implemented the higher-order analysis and since it is considerably more complex than the first-order analysis, we do not give an example of a generating extension, but instead analyze another program-transformation program. The results of our analysis were used by Danvy [Dan92] as part of the proof of his transformation.

Finally we suggest an application of the analysis as part of a general programming language workbench.

Except for shortening the non-terminals and removing a few unreachable or empty rules, we have reproduced the grammars for the first-order analysis exactly as they are produced by our implementation (as described in chapter 7).

In all the examples, the analysis is run with the goal function `specialize-0` and with Val and the abstract argument.

5.1 A Toy Example

To get a feel for the way the analysis works, and to check that it gives the kind of results that we expected, we first consider the small appending program, that we gave the generating extension for in section 3.3.3.

Analyzing this generating extension with the simple first-order analysis gives the grammar in figure 5.1.

We use the following notational conventions in the grammar: residual expressions are enclosed in "syntactic brackets" `[[` and `]]`. Lists of residual definitions are enclosed in curly braces and have the tag `pgm`. We have explicit tags on the syntax, `cst` for constants, `var` for variables, `@` for application, `@-prim` for application of primitives, `if` for conditional, `let` for let-expressions, and `pgm` for programs. All predefined non-terminals (such as Val) are capitalized. We use an infix "." for *cons*. Residual identifiers are written as a pair of a string and a number (corresponding to their internal representation in Similix), whereas procedure names are simply written as the string when they appear as non-terminals, but as a pair of a string and a number when they appear as identifiers. For brevity, we use the {}-notation for nested lists of alternates.

We have shortened some of the non-terminals, but otherwise the grammar appears as it is produced by the analysis.

The start symbol is `v-PROC-spec-0`. The rule for this non-terminal tells us that the residual

```
v-PROC-spec-0

main --> [[ (define main (("z" . -1) . nil)
               (let ("z" . Nat) [var ("z" . -1)]
                   (@-prim "cons" v-PROC-process-expr-0-2 v-PROC-process-expr-0-2))) ]]

s2-PROC-process-expr-0-2 --> {pgm } | s2-PROC-process-expr-0-2

v-PROC-process-expr-0-2 --> Val
                          | [[ (@-prim "cons" [const Val] v-PROC-process-expr-0-2) ]]

v-PROC-spec-0 --> ("scheme.adt" . {pgm main :s2-PROC-process-expr-0-2})
                | ("scheme.adt" . {pgm })
```

Figure 5.1: The grammar for the residual programs corresponding to the program in figure 3.3 with the two first arguments static and the third dynamic.

programs will consist of an expression loading the adt-file with the usual Scheme definitions and one residual procedure (since the rule for `s2-PROC-process-expr` generates the empty language).

The procedure is defined in the rule for the non-terminal `main`. It takes one argument, `z-1` (the internal representation of the identifier is the pair `(z . -1)`). It then renames the argument (this let-expression will actually be post-unfolded), and cons-es two instances of `v-PROC-process-expr-0-2`.

The non-terminal `v-PROC-process-expr-0-2` can be either **Val** or the expression that cons-es a constant onto `v-PROC-process-expr-0-2`.

The grammar looks almost like we wanted it to, except for the occurrence of **Val**. Since **Val** could be anything, this means that anything could be in the residual program.

The problem occurs because the analysis generalizes the arguments to procedures to **Val**. We can get the result that we were looking for by using the analysis that generates non-terminals for the parameters. The resulting grammar is given in figure 5.2.

In this grammar, the analysis was able to retain the information that the Val in the rule for `v-PROC-process-expr-0-2` is actually an instance of the identifier `z`. Since there is only that one possibility, our implementation does not actually generate the rule for the parameter.

5.2 Example: the MP Interpreter

Considering the emphasis on partial evaluation as a compiler generation tool, it is natural that we take an interpreter example. We use the MP interpreter that is distributed with Similix [Bon91b]. This toy interpreter is used to demonstrate compiler generation by partial evaluation. We have chosen it as an example since it was written by someone else without any consideration for the present analysis.

The syntax of MP programs is given in figure 5.3. It is a small imperative language with assignment, conditional and while-statements. Expressions are Lisp-like S-expressions.

The interpreter takes an MP program and a list of values corresponding to the arguments (`pars`) and returns a dummy statement. It uses several auxiliary primitives and global data-structures that are defined in an adt-file. The main structures are the environment and the store, that are both

```
v-PROC-spec-0
main --> [[ (define main (("z" . -1) . nil)
               (let ("z" . Nat) [var ("z" . -1)]
                   (@-prim "cons" v-PROC-process-expr-0-2 v-PROC-process-expr-0-2))) ]]

s2-PROC-process-expr-0-2 --> {pgm } | s2-PROC-process-expr-0-2

v-PROC-process-expr-0-2 --> [[ [var ("z" . Nat)] ]]
                          | [[ (@-prim "cons" [const Val] v-PROC-process-expr-0-2) ]]

v-PROC-spec-0 --> ("scheme.adt" . {pgm main :s2-PROC-process-expr-0-2})
               | ("scheme.adt" . {pgm })
```

Figure 5.2: The grammar with rules for parameters for the residual programs corresponding to the program in figure 3.3 with the two first arguments static and the third dynamic.

```
; P in Program
; B in Block
; C in Command
; E in Expression
; V in Variable
; Cst in Constant
;
; P = (program (pars V1*) (vars V2*) B)
; B = (C*)
; C = (:= V E)
;   | (if E B1 B2)              $ first branch iff exp not ()
;   | (while E B)               $ loop iff Exp not ()
; E = Cst
;   | V
;   | (car E)
;   | (cdr E)
;   | (cons E1 E2)
;   | (atom E)                  $ () iff not atom
;   | (equal E1 E2)             $ () iff not equal
;
```

Figure 5.3: Syntax of MP programs as given in the Similix package.

```
(loadt "scheme.adt")
(loadt "MP-int.adt")

(define (run P value*)
  (let* ((V2* (P->V2* P))
         (env (init-environment (P->V1* P) V2*)))
    (init-store! value* (length V2*))
    (evalBlock (P->B P) env)))

(define (evalBlock B env)
  (if (emptyBlock? B)
      "Finished block"
      (evalCommands (headBlock B) (tailBlock B) env)))

(define (evalCommands C B env)
  (if (emptyBlock? B)
      (evalCommand C env)
      (begin (evalCommand C env)
             (evalCommands (headBlock B) (tailBlock B) env))))

(define (evalCommand C env)
  (cond ((isAssignment? C)
         (update-store! (lookup-env (C-Assignment->V C) env)
                        (evalExpression (C-Assignment->E C) env)))
        ((isConditional? C)
         (if (is-true? (evalExpression (C-Conditional->E C) env))
             (evalBlock (C-Conditional->B1 C) env)
             (evalBlock (C-Conditional->B2 C) env)))
        ((isWhile? C)
         (if (is-true? (evalExpression (C-While->E C) env))
             (begin (evalBlock (C-While->B C) env)
                    (evalCommand C env))
             "Finished loop"))
        (else "Error - unknown command")))

(define (evalExpression E env)
  (cond ((isConstant? E)
         (constant-value E))
        ((isVariable? E)
         (lookup-store (lookup-env (E->V E) env)))
        ((isPrim? E)
         (let ((op (E->operator E)))
           (cond ((is-cons? op)  (cons (evalExpression (E->E1 E) env)
                                       (evalExpression (E->E2 E) env)))
                 ((is-equal? op) (equal? (evalExpression (E->E1 E) env)
                                         (evalExpression (E->E2 E) env)))
                 ((is-car? op)   (car (evalExpression (E->E E) env)))
                 ((is-cdr? op)   (cdr (evalExpression (E->E E) env)))
                 ((is-atom? op)  (atom? (evalExpression (E->E E) env)))
                 (else "Unknown operator"))))
        (else "Unknown expression form")))
```

Figure 5.4: The MP interpreter from Similix.

```
(program (pars x y) (vars out next kn)
  ((:= kn y)

   (while kn
     ((:= next (cons x next))
      (:= kn (cdr kn))))

   (:= out (cons next out))

   (while next
     ((if (cdr (car next))
        ((:= next (cons (cdr (car next)) (cdr next)))

         (while kn
           ((:= next (cons x next))
            (:= kn (cdr kn))))
         (:= out (cons next out)))
       ;else
         ((:= next (cdr next))
          (:= kn (cons '1 kn))))))))
```

Figure 5.5: A program in the MP language

defined as global variables. Thus the final store is the "real" result of interpretation. The adt-file also contains definitions of the syntax destructors.

The two main procedures of the interpreter are `evalCommand` and `evalExpression`, that both dispatch on the syntactic forms and interpret the subterms.

Note that syntax errors will only be reported if there is an attempt to execute the erroneous part of the program.

Figure 5.5 shows a program in the MP language that computes the power function. Since there are no operations on numbers in the language, an integer n is represented by a list of length n.

The intention of the interpreter is, as usual, that the environment is a "compile-time" structure and the store is a "run-time" structure. Thus we would expect all residual programs to access the store directly and not use any environment operations. Similarly, we expect no syntax destructors in the residual programs. Correspondingly, we should be able to remove large parts of the adt-file.

Figure 5.6 shows the compiled version of the power program, obtained by specializing the interpreter with respect to the program. As expected, this compiled program does access the store directly.

By analyzing the generating extension corresponding to the MP interpreter, we get the grammar shown in figure 5.7.

As before, the start symbol is `v-PROC-spec-0`. This time, there are several procedures in the residual programs, and it is more obvious that the grammar gives the residual programs as they look before post-unfolding.

The residual programs now consist of two expressions loading the adt-files, a goal procedure described by non-terminal `run`, and possibly several instances of other procedures, described by non-terminals `evalcommand-0` and `evalcommand-1`. Notice that we can see that all instances of

```
(loadt "MP-int.adt")
(loadt "scheme.adt")
(define (run-0 value*_0)
  (init-store! value*_0 3)
  (update-store! 4 (lookup-store 1))
  (evalcommand-0-1)
  (update-store! 2 (cons (lookup-store 3) (lookup-store 2)))
  (evalcommand-0-2))

(define (evalcommand-0-2)
  (if (is-true? (lookup-store 3))
      (begin (if (is-true? (cdr (car (lookup-store 3))))
                 (begin (update-store! 3 (cons (cdr (car (lookup-store 3)))
                                               (cdr (lookup-store 3))))
                        (evalcommand-0-1)
                        (update-store! 2 (cons (lookup-store 3)
                                               (lookup-store 2))))
                 (begin (update-store! 3 (cdr (lookup-store 3)))
                        (update-store! 4 (cons 1 (lookup-store 4)))))
             (evalcommand-0-2))
      "Finished loop"))

(define (evalcommand-0-1)
  (if (is-true? (lookup-store 4))
      (begin (update-store! 3 (cons (lookup-store 0) (lookup-store 3)))
             (update-store! 4 (cdr (lookup-store 4))) (evalcommand-0-1))
      "Finished loop"))
```

Figure 5.6: Compiled version of the MP program from figure 5.5.

```
evalcommand-0 --> [[ (define evalcommand-0 nil
                        (if (@-prim "is-true?" v-PROC-process-expr-0-5)
                            (begin (v-PROC-process-expr-0-2
                                    . (v-PROC-process-expr-0-4 . nil)))
                            [const "Finished loop"])) ]]

evalcommand-1 --> [[ (define evalcommand-1 nil
                        (if (@-prim "is-true?" v-PROC-process-expr-0-5)
                            v-PROC-process-expr-0-2
                            v-PROC-process-expr-0-2)) ]]

run --> [[ (define run (("value" . -1) . nil)
              (begin ((@-prim "init-store!" [var ("value" . -1)] [const Val])
                      . (v-PROC-process-expr-0-2 . nil)))) ]]

s2-PROC-process-expr-0-2 --> {pgm } | s2-PROC-process-expr-0-3

s2-PROC-process-expr-0-3 --> s2-PROC-process-expr-0-4

s2-PROC-process-expr-0-4 --> {pgm }
                           | {pgm evalcommand-0 :s2-PROC-process-expr-0-5}
                           | {pgm evalcommand-1 :s2-PROC-process-expr-0-5}
                           | s2-PROC-process-expr-0-5

s2-PROC-process-expr-0-5 --> {pgm }

v-PROC-process-expr-0-2 --> [[ [const "Finished block"] ]] |  v-PROC-process-expr-0-3

v-PROC-process-expr-0-3 --> [[ (begin (v-PROC-process-expr-0-4
                                       . (v-PROC-process-expr-0-3 . nil))) ]]
                          |  v-PROC-process-expr-0-4

v-PROC-process-expr-0-4 --> [[ [const "Error - unknown command"] ]]
                          |  [[ (@-prim "update-store!" [const Val]
                                  v-PROC-process-expr-0-5) ]]
                          |  [[ (@ ("evalcommand-0" . Nat) nil) ]]
                          |  [[ (@ ("evalcommand-1" . Nat) nil) ]]

v-PROC-process-expr-0-5 --> [[ [const Val] ]]
 | [[ [const "Unknown expression form"] ]]
 | [[ [const "Unknown operator"] ]]
 | [[ (@-prim "atom?" v-PROC-process-expr-0-5) ]]
 | [[ (@-prim "car" v-PROC-process-expr-0-5) ]]
 | [[ (@-prim "cdr" v-PROC-process-expr-0-5) ]]
 | [[ (@-prim "cons" v-PROC-process-expr-0-5 v-PROC-process-expr-0-5) ]]
 | [[ (@-prim "equal?" v-PROC-process-expr-0-5 v-PROC-process-expr-0-5) ]]
 | [[ (@-prim "lookup-store" [const Val]) ]]

v-PROC-spec-0 --> ("MP-int.adt" . ("scheme.adt" . {pgm run : s2-PROC-process-expr-0-2}))
                | ("MP-int.adt" . ("scheme.adt" . {pgm }))
```

Figure 5.7: Grammar describing the compiled MP programs. **v-PROC-spec-0** is the start symbol.

`evalExpression` must be unfolded at specialization time.

The goal procedure takes one argument and uses it to initialize the store.

The instances of the procedure described by non-terminal `evalcommand-0` take no arguments, test an expression and either return `"Finished loop"` or do a sequence corresponding to `v-PROC-process-expr-0-2` and then `v-PROC-process-expr-0-4`.

The instances of `evalcommand-1` also test an expression, but both branches are described by `v-PROC-process-expr-0-2`.

The non-terminal `v-PROC-process-expr-0-4` corresponds to the actual compiled commands. `evalcommand-0` is a compiled loop, and `evalcommand-1` is a compiled conditional.

The non-terminal `v-PROC-process-expr-0-5` corresponds to the compiled MP-expressions. Even though `Val` represents any value, the form `(const Val)` is more constrained, since it represents a constant in the residual program, the value of which cannot be determined by the analysis. There are two alternates corresponding to syntax errors. The remaining productions correspond to the compilation of well-formed expressions.

The grammar shows us that syntax errors will be reported at run-time rather than at compile-time. This is consistent with the semantics of the interpreter, which may terminate normally on input that does not cause the part with the syntax error to be executed.

It also shows that, as expected, environment lookups have disappeared, whereas store lookups remain. The residual programs do not use any of the syntax destructors, either.

The results of this analysis could be combined with one or more postprocessing steps that could give more specific information automatically. A very simple example for Similix would be a step that could make a specialized "run-time package" of adt files, since the grammar can often tell which of the user-defined primitives are used (or rather, not used) by the residual programs.

5.3 Example: Matching Regular Expressions

In this section we will consider one of the many applications of partial evaluation to pattern matching. The general purpose of these experiments is to show that good, known, pattern-matching algorithms and structures can be obtained automatically by partial evaluation.

The example we consider here is a program that matches a string against a regular expression. When this program is specialized with respect to a regular expression, the residual program has the structure of an automaton. This effect is obtained by binding-time improving a simple regular expression interpreter.

The example was developed at DIKU by Torben Mogensen, Anders Bondorf, and Jesper Jørgensen. It is described in [Bon90].

Figure 5.8 shows the regular expression interpreter of Bondorf, Jørgensen, and Mogensen. Specializing this interpreter with respect to a regular expression gives a residual program that is structured like an automaton. The automaton is usually minimal, but the authors also give an example of a regular expression for which it is not.

The interpreter uses a number of auxiliary procedures, that are all completely static and defined in the adt file. We don't show this here, since it is actually quite long.

The auxiliary `accept-empty` returns true if the regular expression accepts the empty string and false otherwise.

The auxiliary `first` returns the list of all characters that are legal first characters in the strings accepted by the regular expression.

The auxiliary `next` determines the regular expression that we continue with, given that we have already matched the symbol `a`. So if the regular expression is `(a b) ! (c d)`, and we have matched

```
(loadt "scheme.adt")
(loadt "reg-int.adt")

(define (match r s)
  (if (null? s)
      (accept-empty? r)
      (let ((fst (first r)))
        (and (not (null? fst))
             (match1 r (car s) s fst)))))

(define (match1 rex sym str fst)
  (and (not (null? fst))
       (let ((a (car fst)))
         (if (equal? sym a)
             (match (next rex a) (cdr str))
             (match1 rex sym str (cdr fst))))))
```

Figure 5.8: The regular expression interpreter of Bondorf, Jørgensen, and Mogensen

`c`, next returns `d`.

The goal function `match` checks if the string is empty and returns true or false, depending on whether the regular expression accepts empty. If not, it determines the list of legal first symbols and calls `match1`.

The procedure `match1` recurs over the list of first symbols, and returns false if the first symbol in the string is not equal to any of them.

To see if our analysis can determine that the residual program is an automaton, we first need to establish what is meant by saying that a program is an automaton. For this, we first look at a few examples, then we consider the residual program before post-unfolding to establish a firm connection between the call graph and the states and transitions of the automaton. Finally we consider the grammar generated by our analysis.

In figure 5.9, we show two residual programs corresponding to the regular expression interpreter specialized with respect to `((a ! b) c)` and `(((a ! b) c) *)`.

We first note that in pre-processing, Similix adds a new goal procedure to the source program, so that the goal procedure is never called inside the program. So it renames `match` to `match-0`, `match1` to `match1-0` and calls the new goal procedure `match`.

The first residual program has two procedures, corresponding to an instance of the goal function `match` (named `match-0`) and to an instance of `match-0` (named `match-0-1`). But we note that another instance of `match-0` has been unfolded into `match` and the equality tests in different instances of `match1-0` have been unfolded into both the residual procedures.

The second residual program has three residual procedures, and here the first instance of `match-0` has not been unfolded into the instance of `match`, since it is called elsewhere in the program. Also, `match-0-1` corresponds to an accept state, since it returns true if the string is empty.

If we consider the residual programs before post-unfolding, the automata structure becomes clearer.

Each instance of the procedure `match-0` corresponds to a state. It tests if the string is empty, if it is an accept state, it returns true, otherwise it returns false.

```
(loadt "reg-int.adt")
(loadt "scheme.adt")
(define (match-0 s_0)
   (if (null? s_0)
       ()
       (let ([sym_1 (car s_0)])
          (cond
             [(equal? sym_1 'a) (match-0-3 (cdr s_0))]
             [(equal? sym_1 'b) (match-0-3 (cdr s_0))]
             [else ()]))))
(define (match-0-3 s_0)
   (if (null? s_0)
       ()
       (and (equal? (car s_0) 'c) (null? (cdr s_0)) #t)))


(loadt "reg-int.adt")
(loadt "scheme.adt")
(define (match-0 s_0) (match-0-1 s_0))
(define (match-0-1 s_0)
   (if (null? s_0)
       #t
       (let ([sym_1 (car s_0)])
          (cond
             [(equal? sym_1 'a) (match-0-3 (cdr s_0))]
             [(equal? sym_1 'b) (match-0-3 (cdr s_0))]
             [else ()]))))
(define (match-0-3 s_0)
   (cond
      [(null? s_0) ()]
      [(equal? (car s_0) 'c) (match-0-1 (cdr s_0))]
      [else ()]))
```

Figure 5.9: Residual programs for the regular expression interpreter specialized with respect to `((a ! b) c)` and `(((a ! b) c) *)`.

Each instance of the procedure `match1-0` corresponds to a transition. It tests if the first character in the string is equal to "its" symbol, in which case it advances the string and calls an instance of `match-0` (a state). If not, it calls an instance of `match1-0` with the same symbol (try another transition).

The goal function is a pointer to the start state.

When we use the version of our analysis that generates non-terminals for parameters on the example, we get the grammar in figure 5.10.

The start symbol is `v-PROC-spec-0`. The grammar tells us that the residual programs (before post-unfolding of corridor calls and redundant let-expressions) will consist of a goal procedure `match` (with some number attached) and several instances of the procedures `match-0` and `match1-0` (again we know that Similix will add an extra number at the end to distinguish the instances).

The rule for the instances of procedure `match` simply says that it takes one argument and calls one of the instances of `match-0` with this arguments (there are two alternates in the rule for `v-PROC-spec-pcall-0-2` but we can determine which of them is used because the variables names are different and the partial evaluator will not generate a mal-formed program from a well-formed one).

The rule for the instances of `match-0` says that each of these procedures will test if the string is empty and in that case return a constant (which must be either true or false, since the source program returns either true or false). If not, it takes the head of the string and calls one of the instances of `match1-0`.

This corresponds to our definition of a state.

The rule for the instances of `match1-0` gives that each of these procedures tests the head of the string against a constant and if it matches, calls an instance of `match-0` with the tail. If it does not match, it either calls another instance of `match1-0` with the same head and tail or returns false.

This corresponds to our definition of a transition on the constant.

Note that we distinguish between alternates based on a meta-level reasoning about well-definedness of variables. We could avoid this by distinguishing between call-sites, that is, by generating different non-terminals for calls to the same procedure from different call-sites.

The grammar also shows that none of the auxiliaries in `reg-exp.adt` are used in the residual program, so we can avoid loading this adt-file.

5.4 Example: the CPS Transformer

In this section we give an application of the higher-order analysis to a CPS-transformer, showing that the analysis also applies to other program-transformation programs than generating extensions. Since generating extensions are too voluminous to reproduce in the text, the examples so far have been missing an aspect, since we have not actually shown the program that we have been analyzing. This can be misleading since the amount of work that the analysis has to do is not explicit, and on the other hand, the relatively simple connection between the programs we are analyzing and the resulting grammars is not explicit, either. Since we have not implemented the higher-order analysis, we choose a direct example to illustrate the algorithm.

The CPS transformer can be viewed as a two-level term, with ordinary lambda terms and "syntax constructors", that might be interpreted in a variety of ways. It is not surprising that the analysis will also work on this kind of program, since generating extensions are also two-level terms with some extra machinery for handling polyvariance. Thus the current example also points towards possible applications of our analysis in the two-level framework of Nielson and Nielson [NN88].

```
PAR-v-PROC-process-expr-0-4-0 --> [[ [var ("str" . Nat)] ]]

PAR-v-PROC-process-expr-0-4-1 --> [[ [var ("sym" . Nat)] ]]

PAR-v-PROC-spec-pcall-0-2-1 --> (("s" . -1) . nil)

PAR-v-PROC-spec-pcall-0-2-2 --> ([[ (@-prim "cdr" [var ("str" . Nat)]) ]] . nil)
                              | ([[ [var ("s" . -1)] ]] . nil)

PAR-v-PROC-spec-pcall-0-2-4 --> [[ [var ("s" . -1)] ]]

match --> [[ (define match (("s" . -1) . nil) v-PROC-spec-pcall-0-2) ]]

match-0 --> [[ (define match-0 PAR-v-PROC-spec-pcall-0-2-1
                    (let ("s" . Nat) PAR-v-PROC-spec-pcall-0-2-4
                        (if (@-prim "null?" [var ("s" . Nat)])
                         [cst Val]
                         {[cst nil]
                          | (let ("sym" . Nat) (@-prim "car" [var ("s" . Nat)])
                                (let ("str" . Nat) [var ("s" . Nat)]
                                    v-PROC-process-expr-0-4))}))) ]]

match1-0 --> [[ (define match1-0 (("sym" . -2) . (("str" . -1) . nil))
                    (let ("sym" . Nat) [var ("sym" . -2)]
                        (let ("str" . Nat) [var ("str" . -1)]
                            (if (@-prim "equal?" [var ("sym" . Nat)] [cst Val])
                                v-PROC-spec-pcall-0-2
                                (let ("sym" . Nat) [var ("sym" . Nat)]
                                    (let ("str" . Nat) [var ("str" . Nat)]
                                        v-PROC-process-expr-0-4)))))) ]]

s2-PROC-process-expr-0-4 --> {pgm } | {pgm match1-0 :s2-PROC-process-expr-0-4}
                           | s2-PROC-spec-pcall-0-2

s2-PROC-spec-pcall-0-2 --> {pgm } | {pgm match-0}
                         | {pgm match-0 :s2-PROC-process-expr-0-4}
                         | {pgm match-0 :s2-PROC-spec-pcall-0-2}
                         | s2-PROC-spec-pcall-0-2

v-PROC-process-expr-0-4 --> [[ [cst nil] ]]
                          | [[ (@ ("match1-0" . Nat)
                                  (PAR-v-PROC-process-expr-0-4-1
                                   . (PAR-v-PROC-process-expr-0-4-0 . nil))) ]]

v-PROC-spec-pcall-0-2 --> [[ (@ ("match-0" . Nat) PAR-v-PROC-spec-pcall-0-2-2) ]]

v-PROC-spec-0 --> ("reg-int.adt"
                    . ("scheme.adt" . {pgm match :s2-PROC-spec-pcall-0-2}))
               | ("reg-int.adt" . ("scheme.adt" . {pgm }))
```

Figure 5.10: The grammar obtained by analyzing the generating extension of the regular expression matcher. As before, v-PROC-spec-0 is the start symbol.

This application was motivated and used independently of the present work [Dan93].

Continuation-passing style (CPS) is a restricted form of lambda terms, that is often used by implementors [Ste78, App92]. CPS terms are evaluation-order independent and all calls are tail-calls. Every lambda term can be transformed into an equivalent CPS term, as described, *e.g.*, by Plotkin [Plo75]. The BNF of CPS terms is given in figure 5.12.

```
(define (transform e)
  (make-Lam 'k (cpst e l1:(lambda (v)
                            (make-App (make-Ide 'k) v)))))

(define (cpst e c)
  (variant-case e
    [(Cst v) (c (make-Cst v))]
    [(Ide x) (c (make-Ide x))]
    [(Lam x e)
     (c (make-Lam x (make-Lam 'k (cpst e c1:(lambda (v)
                                              (make-App (make-Ide 'k) v))))))]
    [(App e0 e1)
     (cpst e0 c2:(lambda (v0)
                   (cpst e1 c3:(lambda (v1)
                                 (make-App (make-App v0 v1)
                                           (let ([v (gensym! "v")])
                                             (make-Lam v (c (make-Ide v))))))))))]))
```

Figure 5.11: CPS transformer

We consider here a CPS transformer from Danvy and Filinski [DF92], which uses explicit syntax constructors to construct the CPS term. The transformer relies heavily on higher order programming. We concentrate on the essential parts of the transformer and assume that the source term is always well-formed. In [Dan92], Danvy needs the BNF of the terms produced by the transformer. Our analysis produces such a BNF automatically.

In the extended version [Dan93], Danvy factorizes the CPS transformation in several stages and repeatedly uses our analysis to establish the intermediate BNFs of the intermediate languages.

The transformer is shown in figure 5.11. It is decorated with program points at the lambda abstractions.

The construction `variant-case` is macro-expanded into a series of conditional expressions that de-structure the `e`. We simply assume that the tests return the top element of the domain of right hand sides, independently of arguments, *i.e.*, that the `variant-case` cannot be resolved. The primitives used by the CPS transformer are slightly different from the ones used by Similix, but the

```
E ::= (lambda (k) S)
S ::= (@ k T) | (@ T T (lambda (I) S))
T ::= C | I | (lambda (I) (lambda (k) S))
```

Figure 5.12: BNF of canonical CPS terms

transform	→	(Lam k cpst)
transform-c	→	**Val**
cpst	→	**t1|c1|c2|c3|cpst**
cpst-c	→	**transform-c|Val**
cpst-c	→	ε
t1	→	(App (Ide k) **t1-v**)
t1-v	→	(Cst **Val**)|(Ide **Val**)|(Lam **Val** (Lam k **cpst**))|(Ide v-**Nat**)
c1	→	(App k **c1-v**)
c1-v	→	(Cst **Val**)|(Ide **Val**)|(Lam **Val** (Lam k **cpst**))|(Ide v-**Nat**)
c2	→	**cpst**
c2-v0	→	(Cst **Val**)|(Ide **Val**)|(Lam **Val** (Lam k **cpst**))|(Ide v-**Nat**)
c3	→	(App (App **c2-v0 c3-v1**) (Lam v-**Nat** {**t1** | **c1** | **c2** | **c3**}))
c3-v1	→	(Cst **Val**)|(Ide **Val**)|(Lam **Val** (Lam k **cpst**))|(Ide v-**Nat**)

Figure 5.13: The result of analyzing the CPS transformer

transform	→	(Lam k cpst)
cpst	→	**t1|c1|c3**
t1	→	(App (Ide k) **t1-v**)
t1-v	→	(Cst **Val**)|(Ide **Val**)|(Lam **Val** (Lam k **cpst**))|(Ide v-**Nat**)
c1	→	(App k **c1-v**)
c3	→	(App (App **t1-v t1-v**) (Lam v-**Nat** {**t1** | **c1** | **cpst** | **c3**}))

Figure 5.14: The result of analyzing the CPS transformer, simplified

difference is only cosmetic. We let the abstract version of `gensym` return the pair of its argument and the top element in the domain of numbers. The syntax constructors `make-Cst`, `make-Ide`, `make-Lam`, and `make-App` clearly return values in the domain *Res-Expr*. If `make-Cst` is given an argument $(c^*, [\ldots, a_i, \ldots], \gamma)$, we define it to return the value $([], [\ldots, inRes\text{-}Expr(inRes\text{-}Cst(a_i)), \ldots], \gamma)$. Note that since we have assumed that it would be an error to apply `make-Cst` to a closure, if c^* is not empty, the result is an error, which we assume is always included in the abstract value. This definition of an abstract `make-Cst` is clearly safe. We define the other syntax constructors similarly.

The grammar produced by the analysis on the CPS transformer is given in figure 5.13. For non-terminals we use simply the procedure names and program points as given in figure 5.11. The non-terminals for parameters are the procedure non-terminals concatenated with the parameter name. The non-terminals are in bold-face. The start symbol is **transform**, since `transform` is the start procedure.

As an example of how the grammar is constructed, note for example that the result of the body of `transform` quite clearly is $(inRes\text{-}Expr(inRes\text{-}Lam(inRes\text{-}Ide(k), \text{cpst})))$, or, in a more readable form (Lam k **cpst**) (using the same notation as for the input syntax). The result of the body of cpst is clearly the result of applying the higher-order argument c or the result of applying cpst, and so on. The grammar is quite verbose; clearly bigger than what we want. But several of the productions are in fact identical (for example the ones corresponding to the arguments of the lambda abstractions, since they are all passed through the parameter c and potentially applied at the same application sites).

If we eliminate these redundant productions and the unreachable productions for **transform-c**,

cpst-c, and **cpst-c**, and also remove the circular rules **cpst** → **cpst** and **cpst** → **c2**, **c2** → **cpst**, we get the simplified grammar in figure 5.14. If we then eliminate the non-terminals **t1**, **c1**, **c3**, and generalize slightly to forget about the specific restrictions of identifiers (such as generalizing v-*Nat* to just identifier), we obtain the usual BNF of CPS terms.

5.5 Generation of "Super-Combinators" by Pre-Analysis

Since our analysis gives us a grammar of residual programs, we can use this to improve compilation by partial evaluation. The grammars tell us which combinations of residual terms are possible. In that sense they form a kind of "super-combinators" of residual terms. An interpreter for that restricted subset of the residual language is likely to be simpler than an interpreter for the entire language. If we name each of the combinators, we can use the information from the grammars to modify the generating extension, so that it produces these names instead of residual syntax. We could also design a partial evaluator that will automatically specialize an interpreter for the residual language to the subset defined by a grammar. This would complete an automatic system of compiler generation from definitional interpreters for high-level languages to a customized mid-level combinator form.

As an example of how this would work, we can consider the MP interpreter example above. The rule for `v-PROC-process-expr-0-5` gives a short, but exhaustive list of the value-typed operations of the residual programs and also defined how they may be nested. Similarly, the rule for `evalcommand-0` defines one loop-construction and the rule for `evalcommand-1` defines one conditional. These are the only control structures that the residual programs can use.

Chapter 6

Formalization and Safety of the Analysis

In this chapter, we will prove the safety of the first-order and the higher-order grammar-generating semantics, as given in chapter 4.

In both cases, we will first formalize safety as a relation between the standard values and grammars that captures the notion of "the standard value is in the language generated by the abstract value".

We prove that the relation holds between the standard and the grammar-generating semantics by factorizing the two semantics with a common core and two different interpretations. The relation between the domains is extended to a relation between the interpretations using logical relations. Safety is then proven by proving that the relation holds between the interpretations, following the technique outlined in section 2.2.7.

We do not present a formal proof of termination of the algorithms, but only outline such a proof in both cases.

6.1 Safety of the First-Order Grammar-Generating Interpretation

We first give the core, which is obtained by observing which parts of the semantics are the same in both cases. Generally, we seek to leave as much as possible in the core, since that will likely minimize the proof.

Then we give a standard and a grammar-generating interpretation of the core and define a relation between the standard and the abstract domains. We then prove that this relation holds between the two interpretations.

6.1.1 The core for the first-order language

The domain of values is constructed quite differently in the two semantics, and the procedure environment has an extra component in the grammar-generating semantics. So these domains are only specified in the interpretations, and any operations on them have to be handled by combinators, that are also specified in the interpretations.

Corresponding to the different procedure environments, the denotation of procedures is also different, since the grammar-generating interpretation has memo-function lookups and extensions. If we encapsulate this management in the procedure itself and in procedure lookup, actual application can be the same in the two interpretations.

Domains:

$v \in Val$ $\quad$ $\pi \in Proc\text{-}Template = Penv \to Proc$

$\rho \in Env = (\mathrm{Id} \to Val)_\perp$ $\quad$ $\sigma \in Store = Val \times Val$

$\varphi \in Penv$ $\quad$ $(v, \sigma) \in Result = Val \times Store$

$f \in Proc = Val^* \to Store \to Result$ $\quad$ $\theta \in Thunk = Store \to Result$

Valuation functions:

$\mathcal{P} : \mathrm{Pgm} \to \text{Proc-Name} \to Val^* \to Val$

$\mathcal{P}[\![((\texttt{define } (\mathtt{F}_1\ \mathtt{I}_{1_1} \ \ldots\ \mathtt{I}_{n_1})\ \mathtt{E}_1) \ \ldots\ (\texttt{define } (\mathtt{F}_m\ \mathtt{I}_{1_m} \ \ldots\ \mathtt{I}_{n_m})\ \mathtt{E}_m))]\!]\,\mathtt{F}\,v^* =$

$\quad$ *let* $\varphi = \textbf{mk-penv}\,[\mathtt{F}_1, \ldots, \mathtt{F}_m]\,[\lambda\,\varphi\,[v_1, \ldots, v_{n_1}].\ \mathcal{E}[\![\mathtt{E}_1]\!][\mathtt{I}_1 \mapsto v_1, \ldots, \mathtt{I}_{n_1} \mapsto v_{n_1}]\lambda\, i.\ \perp_{Val},\ \varphi, \ldots,$

$\qquad\qquad \lambda\,\varphi\,[v_1, \ldots, v_{n_m}].\ \mathcal{E}[\![\mathtt{E}_m]\!][\mathtt{I}_1 \mapsto v_1, \ldots, \mathtt{I}_{n_m} \mapsto v_{n_m}]\lambda\, i.\ \perp_{Val},\ \varphi]$

$\quad$ *in* $((\textbf{lookup-proc}\,\mathtt{F}\,\varphi\,v^*\,\sigma_{init})\ \downarrow\ 1)$

$\mathcal{E} : \mathrm{Expr} \to Env \to Penv \to Store \to Result$

$\mathcal{E}[\![\mathtt{C}]\!]\,\rho\,\varphi\,\sigma = (\mathcal{C}[\![\mathtt{C}]\!], \sigma)$

$\mathcal{E}[\![\mathtt{I}]\!]\,\rho\,\varphi\,\sigma = (\rho\,\mathtt{I}, \sigma)$

$\mathcal{E}[\![(\texttt{if } \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!]\,\rho\,\varphi\,\sigma = \textbf{cond}(\mathcal{E}[\![\mathtt{E}_0]\!]\,\rho\,\varphi)(\mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\varphi)(\mathcal{E}[\![\mathtt{E}_2]\!]\,\rho\,\varphi)\,\sigma$

$\mathcal{E}[\![(\mathtt{O}\ \mathtt{E}_1\ \ldots\ \mathtt{E}_n)]\!]\,\rho\,\varphi\,\sigma =$ *let* $(v_1, \sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\varphi\,\sigma$

$\qquad$ *in* $\ldots$ *let* $(v_n, \sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1}$

$\qquad\qquad$ *in* $\mathcal{O}[\![\mathtt{O}]\!]\,[v_1, \ldots, v_n]\,\sigma_n$

$\mathcal{E}[\![(\mathtt{F}\ \mathtt{E}_1\ \ldots\ \mathtt{E}_n)]\!]\,\rho\,\varphi\,\sigma =$ *let* $f = (\textbf{lookup-proc}\,\mathtt{F}\,\varphi)$

$\qquad$ *in* *let* $(v_1, \sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\varphi\,\sigma$

$\qquad\quad$ *in* $\ldots$ *let* $(v_n, \sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1}$

$\qquad\qquad$ *in* $f\,[v_1, \ldots, v_n]\,\sigma_n$

$\mathcal{E}[\![(\texttt{let } ((\mathtt{I}\ \mathtt{E}_1))\ \mathtt{E}_2)]\!]\,\rho\,\varphi\,\sigma = \textit{let}\ (v, \sigma') = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\varphi\,\sigma\ \textit{in}\ \mathcal{E}[\![\mathtt{E}_2]\!]\,[\mathtt{I} \mapsto v]\rho\,\varphi\,\sigma'$

$\mathcal{O} : \mathrm{Op} \to Val^* \to Store \to Result$

$\mathcal{O}[\![\textbf{cons}]\!]\,[v_1, v_2]\,\sigma = (\textbf{cons}\,v_1\,v_2, \sigma)$

$\mathcal{O}[\![\textbf{car}]\!]\,[v]\,\sigma = (\textbf{car}\,v, \sigma)$

$\ldots$

$\mathcal{O}[\![\textbf{\_sim-build-primop1}]\!]\,[v_1, v_2]\,\sigma = (\textbf{build-primop1}\,v_1\,v_2, \sigma)$

$\ldots$

$\mathcal{O}[\![\textbf{\_sim-generate-proc-name!}]\!]\,[v_1, v_2]\,\sigma = \textbf{gen-proc-name}\,v_1\,v_2\,\sigma$

where

$\sigma_{init} : Store$

mk-penv : Proc-Name* $\to$ *Proc-Template** $\to$ *Penv*

cond : *Thunk* $\to$ *Thunk* $\to$ *Thunk* $\to$ *Thunk*

lookup-proc : Proc-Name $\to$ *Penv* $\to$ *Proc*

cons : *Val* $\to$ *Val* $\to$ *Val*

car : *Val* $\to$ *Val*

build-primop1 : *Val* $\to$ *Val* $\to$ *Val*

gen-proc-name : *Val* $\to$ *Val* $\to$ *Store* $\to$ *Result*

Figure 6.1: Core for the first-order language

Since values are different in the two semantics, treatment for each of the primitive operations must be specified in the interpretations. The valuation function $\mathcal{C}$ is also part of the interpretation[1].

Based on these considerations, we get the core semantics of the first-order language given in figure 6.1. We use boldface to indicate combinators.

In the valuation function $\mathcal{P}$, procedure environments are defined with the combinator **mk-penv**.

In the $\mathcal{E}$ valuation function, the standard and the grammar-generating semantics for the first-order language differ in the treatment of the conditional and in the treatment of procedure application.

The denotation of conditionals are defined using the combinator **cond**. Looking up procedure names in the resulting procedure environment is abstracted in the combinator **lookup-proc**. This is sufficient to capture the differences.

For convenience, we have introduced the new domain *Thunk* for the results of the partial applications of the valuation function in the conditional, and the domain *Proc-Template* for the arguments of **mk-penv**.

6.1.2 The standard interpretation

Domains:

$v \in Val_{std} = (Nat + Bool + Str + Pair_{std} + Res\text{-}Expr)$

$(v_1, v_2) \in Pair_{std} = Val_{std} \times Val_{std}$

$\varphi \in Penv_{std} = (\text{Proc-Name} \rightarrow Proc_{std})_\perp$

Combinators:

$\sigma_{init\ std} = (Nil, Nil)$

$\textbf{mk-penv}_{std} = \lambda\,[F_1, \ldots, F_n]\,[\pi_1, \ldots, \pi_n].\ fix\ \lambda\,\varphi.\,[F_1 \mapsto \pi_1\,\varphi, \ldots, F_n \mapsto \pi_n\,\varphi]\lambda\,f.\ \perp_{Proc_{std}}$

$\textbf{cond}_{std} = \lambda\,\theta_0\,\theta_1\,\theta_2\,\sigma.\ let\ (Bool(b), \sigma_0) = (\theta_0\,\sigma)\ in\ (b \rightarrow \theta_1\,\sigma_0 \,[]\, \theta_2\,\sigma_0)$

$\textbf{lookup-proc}_{std} = \lambda\,F\,\varphi.\ (\varphi F)$

$\textbf{cons}_{std} = \lambda\,v_1\,v_2.\ in Pair(v_1, v_2)$

$\textbf{car}_{std} = \lambda\,v.\ let\ Pair(v_1, v_2) = v\ in\ v_1$

$\textbf{build-primop1}_{std} = \lambda\,v_1\,v_2.\ in Res\text{-}Expr(primop1, v_1, v_2)$

$\textbf{gen-proc-name}_{std} = \lambda\,v_1\,v_2\,(\sigma_1, \sigma_2).\ cases\ (seenb4?\,v_1\,v_2\,\sigma_1)\ of$

$\quad isEntry(v_n) \rightarrow (in Pair(v_n, True), (\sigma_1, \sigma_2))$

$\quad [] isUnit() \rightarrow let\ v_n = gen\text{-}new\text{-}name\,v_1\,v_2\,\sigma_1$

$\quad\quad in\ (in Pair(v_n, False), ([v_1, v_2, v_n] :: \sigma_1, \sigma_2))$

The valuation function for constants:

$\mathcal{C} : \text{Cst} \rightarrow Val_{std}$

$\mathcal{C}[\![\#t]\!] = in Bool(true)$

Figure 6.2: Standard interpretation of the first-order language

The standard interpretation of the first-order language is shown in figure 6.2.

We call this interpretation *std* and put this as a subscript on the domains and combinators in the interpretation.

[1] Since we have the standard values as part of the abstract domain, we can actually let it return the same value, just injected into the different domains.

To model the standard semantics, the standard interpretation of the domain of values is a coalesced sum of the flatly ordered domains of naturals, booleans, strings, *etc.* The standard interpretation of the domain of procedure environments is the domain of functions from procedure names to procedures.

We assume that $Bool = \{true, false\}_\perp$ and that $True = inBool(true)$.

The interpretation of the combinator **cond** is the usual strict conditional, that determines the value of the test and selects one of the alternates. The interpretation of the combinator **lookup-proc** applies the functional procedure environment to the given procedure name.

The combinator $\textbf{mk-penv}_{std}$ takes two arguments. The first is a list of function names. The second is a list of procedure templates, that given a procedure environment will produce a procedure. It builds a procedure environment by binding each of the function names to the result of applying the corresponding procedure template to the procedure environment. It uses *fix* over the procedure environment to get a procedure environment where all the procedures are mutually recursively defined.

Lemma 1 *The interpretation std of the core in figure 6.1 gives the standard semantics of figure 4.4.*

Proof: This is easily seen by unfolding the definitions of domains and combinators. □

6.1.3 The grammar-generating interpretation

Figure 6.3 gives the grammar-generating interpretation of the first-order language. We call this interpretation γ and use this as a subscript as before.

The domain of values is interpreted as a term (a "right hand side") and a grammar, as described in section 4.2.1. The domain of procedure environments is interpreted as $Template_\gamma \times CEnv$. $Template_\gamma$ is a higher-order procedure environment returning functions from constant procedure environments to procedures. *CEnv* is the memo-function.

The interpretation of the combinator **cond** joins the results of the two branches, except in the case where the abstract value happens to be either the terminal **True** or the terminal **False**, indicating that the test is really redundant.

The combinator **mk-penv** takes a list of procedure names and a list of procedure templates and returns a procedure environment. This is done by fixing over the template part and initializing the memo-function to be empty. The auxiliary $mk\text{-}proc_\gamma$ is applied to the procedure template to add generation of non-terminals and the fixed point and extension of the memo-function (this auxiliary is the same as in figure 4.10).

Lemma 2 *The interpretation γ of the core in figure 6.1 gives the grammar-generating semantics of figure 4.10.*

Proof: This is easily seen by unfolding the definitions of domains and combinators. □

6.1.4 Safety of the first-order grammar-generating semantics with respect to the standard semantics

We prove safety of the analysis using logical relations.

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): $v \in L(\alpha, \gamma)$.

Domains:
$v, (\alpha, \gamma) \in Val_\gamma = Rhs \times Grammar_\gamma$
$\gamma \in Grammar_\gamma = (Non\text{-}Terminal \times Rhs)^*$
$Non\text{-}Terminal = Non\text{-}Term\text{-}Proc + Non\text{-}Term\text{-}Resid$
$\alpha, [a_1, \ldots, a_n] \in Rhs = Alternate_\gamma{}^*$
$a \in Alternate_\gamma = (Nat^\top + Bool^\top + Str^\top + Pair^\top + Res\text{-}Expr_\gamma + Non\text{-}Term\text{-}Proc + Non\text{-}Term\text{-}Resid)^\top$
$n \in Non\text{-}Term\text{-}Proc = Str$
$n \in Non\text{-}Term\text{-}Resid = Str$
$\varphi \in Penv_\gamma = Template_\gamma \times CEnv$
$\tau \in Template_\gamma = (\text{Proc-Name} \rightarrow CEnv \rightarrow Proc_\gamma)_\perp$
$\varphi_c \in CEnv = \text{Proc-Name} \rightarrow CProc$
$f \in CProc = (\{\lambda\, v^*\, \sigma.\, r \mid r \in Result\} \cup \{undef_{CProc}\})_\perp$

Combinators:
$\sigma_{init_\gamma} = (([Nil], \gamma_{empty}), ([Nil], \gamma_{empty}))$
$\textbf{mk-penv}_\gamma = \lambda\, [F_1, \ldots, F_n]\, [\pi_1, \ldots, \pi_n].$
$\quad (fix\ \lambda\, \tau.\, [F_1 \mapsto mk\text{-}proc_\gamma\ F_1\ \pi_1\ \tau, \ldots,$
$\quad\quad F_n \mapsto mk\text{-}proc_\gamma\ F_n\ \pi_n\ \tau]\lambda\, f.\ \lambda\, \varphi_c.\ \perp_{Proc_\gamma}, (\lambda\, F.\ undef_{CProc}))$
where
$mk\text{-}proc_\gamma = \lambda\, F\, \pi\, \tau.$
$\quad \lambda\, \varphi_c.\ \lambda\, v^* \sigma.\ let\ \alpha' = mk\text{-}non\text{-}terminal\ F\ \sigma$
$\quad\quad in\ let\ r_\gamma = fix\ \lambda\, r_\gamma.\ \pi\, (\tau, ([F \mapsto \lambda\, v^* \sigma.\ (abstract\text{-}res\ \alpha'\ r_\gamma)]\varphi_c))$
$\quad\quad\quad (generalize\text{-}args\ v^*)\, (generalize\text{-}store\ \sigma)$
$\quad\quad in\ (abstract\text{-}res\ \alpha'\ r_\gamma)$
where
$abstract\text{-}res = \lambda\, [n_0, n_1, n_2].\ \lambda\, ((\alpha_0, \gamma_0), ((\alpha_1, \gamma_1), (\alpha_2, \gamma_2))).$
$\quad ((n_0, \{(n_0, \alpha_0)\} \cup \gamma_0), ((n_1, \{(n_1, \alpha_1)\} \cup \gamma_1), (n_2, \{(n_2, \alpha_2)\} \cup \gamma_2)))$
$\textbf{cond}_\gamma = \lambda\, \theta_0\, \theta_1\, \theta_2\, \sigma.\ let\ (([a_1, \ldots, a_n], \gamma), \sigma_0) = (\theta_0\, \sigma)$
$\quad in\ ([a_1, \ldots, a_n] = [\text{True}]) \rightarrow (\theta_1\, \sigma_0)$
$\quad\quad \| \ ([a_1, \ldots, a_n] = [\text{False}]) \rightarrow (\theta_2\, \sigma_0)$
$\quad\quad\quad \| \ join(\theta_1\, \sigma_0)\, (\theta_2\, \sigma_0)$
where $join = \lambda\, (\alpha, \gamma)\, (\alpha', \gamma').\ (\alpha \cup \alpha', \gamma \cup \gamma')$
$\textbf{lookup-proc}_\gamma = \lambda\, F\, (\tau, \varphi_c).\ ((\varphi_c\, F) = undef_{CProc}) \rightarrow (\tau\, F\, \varphi_c) \,\|\, \varphi_c F$
$\textbf{cons}_\gamma = \lambda\, (\alpha, \gamma)\, (\alpha', \gamma').\ ([\text{Val}], \gamma_{empty})$
$\textbf{car}_\gamma = \lambda\, (\alpha, \gamma).\ ([\text{Val}], \gamma_{empty})$
$\textbf{build-primop1}_\gamma = \lambda\, ([\ldots, a_i, \ldots], \gamma)\, ([\ldots, a'_j, \ldots], \gamma').\ ([\ldots, in Res\text{-}Expr(primop1, a_i, a'_j), \ldots], \gamma \cup \gamma')$
$\textbf{gen-proc-name}_\gamma = \lambda\, (\alpha, \gamma)\, (\alpha', \gamma')\, \sigma.\ let\ a = gen\text{-}new\text{-}name_\gamma\, (\alpha, \gamma)$
$\quad in\ ((([inPair(inPair(a, \textbf{Nat}), [\textbf{Bool}])]), \gamma \cup \gamma'), \sigma)$

The valuation function for constants:
$C : \text{Cst} \rightarrow Val_\gamma$
$C[\![\#t]\!] = ([\text{True}], \gamma_{empty})$
...

Figure 6.3: Grammar-generating interpretation of the first-order language

$$
\begin{array}{lcl}
\mathcal{P}_{Val}(v,(\alpha,\gamma)) & = & v \in L(\alpha,\gamma) \\
\mathcal{P}_{Store}((v,v'),(v_\gamma,v'_\gamma)) & = & \mathcal{P}_{Val}(v,v_\gamma) \wedge \mathcal{P}_{Val}(v',v'_\gamma) \\
\mathcal{P}_{Result}((v,\sigma),(v_\gamma,\sigma_\gamma)) & = & \mathcal{P}_{Val}(v,v_\gamma) \wedge \mathcal{P}_{Store}(\sigma,\sigma_\gamma) \\
\mathcal{P}_{Proc}(f,f_\gamma) & = & \forall v^* \in Val^*_{std}, \sigma \in Store_{std}, v^*_\gamma \in Val^*_\gamma, \sigma_\gamma \in Store_\gamma : \\
 & & \mathcal{P}_{Val^*}(v^*,v^*_\gamma) \wedge \mathcal{P}_{Store}(\sigma,\sigma_\gamma) \Rightarrow \mathcal{P}_{Result}(f\, v^*\sigma, f_\gamma\, v^*_\gamma\sigma_\gamma) \\
\mathcal{P}_{Penv}(\varphi,(\tau,\varphi_c)) & = & Q_{CEnv}(\varphi,\varphi_c) \wedge \forall \mathrm{F} \in \text{Proc-Name} : \mathcal{P}_{Proc}(\varphi\,\mathrm{F}, \tau\,\mathrm{F}\,\varphi_c) \\
Q_{CEnv}(\varphi,\varphi_c) & = & \forall \mathrm{F} \in \text{Proc-Name} : \varphi_c\,\mathrm{F} = undef_{CProc} \vee \mathcal{P}_{Proc}(\varphi\,\mathrm{F}, \varphi_c\,\mathrm{F})
\end{array}
$$

Figure 6.4: Relations between the standard and the abstract domains for the first-order language

Remember that we assume that the least element of the domain of languages contains all the error output.

The relation between the standard and the abstract domains of procedure environments is intuitively that they take the same procedure name to related procedures. But since the abstract procedure environment is not a function but a tuple with a template of an environment and a memo-function, this needs to be phrased a little carefully.

We subdivide the problem by defining an auxiliary relation between the standard procedure environment and the memo-function. The relation on procedure environments is then that the auxiliary relation holds and that a procedure found in the standard environment must be related to the procedure found by looking up the same name in the abstract environment and applying the result to the memo-function.

The auxiliary relation states that for every procedure name defined in the standard environment, the same name is either undefined in the memo-function or it is bound to a matching procedure.

From these definitions of the relations on the domains specified in the interpretations, the relation is extended in the usual way to the compound domains.

The relation $\mathcal{P}$ is show in figure 6.4. We do not write out the relation on all the compound domains, but give some common examples.

Proof of safety for the first-order grammar-generating interpretation

To prove that the grammar-generating semantics is safe with respect to the standard semantics, we need to prove, for each combinator, that its two interpretations fulfill the relation induced by the type of the combinator. So for example for **mk-penv**, we want to prove that $\mathcal{P}_{Proc\text{-}Name^* \to Proc\text{-}Template^* \to Penv}(\mathbf{mk\text{-}penv}_{std}, \mathbf{mk\text{-}penv}_\gamma)$ holds, that is, for all $[\mathrm{F}_1,\ldots,\mathrm{F}_n]$ in Proc-Name* and $([\pi_1,\ldots,\pi_n],[\pi_{1\gamma},\ldots,\pi_{n\gamma}])$ in $Proc\text{-}Template^*_{std} \times Proc\text{-}Template^*_\gamma$ we have $\mathcal{P}_{Proc\text{-}Template}(\pi_i,\pi_{i\gamma})$ for $i = 1,\ldots,n$ implies
$\mathcal{P}_{Penv}(\mathbf{mk\text{-}penv}_{std}[\mathrm{F}_1,\ldots,\mathrm{F}_n]\,[\pi_1,\ldots,\pi_n], \mathbf{mk\text{-}penv}_\gamma[\mathrm{F}_1,\ldots,\mathrm{F}_n]\,[\pi_{1\gamma},\ldots,\pi_{n\gamma}])$.

Since we need to use fixed point induction, we need the relation we have defined to be an inclusive predicate over the two domains [2]. Fortunately, this holds trivially, as outlined:
$\mathcal{P}_{Val}$: since Val_{std} is flat and L is monotonous, this is inclusive.
$\mathcal{P}_{Store}$: Since $Store$ is $Val \times Val$, inclusivity of $\mathcal{P}_{Store}$ follows from inclusivity of $\mathcal{P}_{Val}$.
$\mathcal{P}_{Result}$: Inclusivity of $\mathcal{P}_{Result}$ follows in the same way from inclusivity of $\mathcal{P}_{Val}$ and $\mathcal{P}_{Store}$.
$\mathcal{P}_{Proc}$: Inclusivity of $\mathcal{P}_{Proc}$ clearly follows from inclusivity of $\mathcal{P}_{Val}$, *etc.*
Q_{CEnv}: Inclusivity of Q_{CEnv}follows from inclusivity of $\mathcal{P}_{Proc}$, equality, and disjunction.

[2] Sometimes also called an "admissible predicate".

$\mathcal{P}_{Penv}$: Inclusivity of $\mathcal{P}_{Penv}$ follows from inclusivity of $\mathcal{P}_{Proc}$, Q_{CEnv}, and conjunction.
and so on.

We can now establish that the two interpretations are related:

Theorem 2 *The relation $\mathcal{P}$ holds between std and γ.*

Proof: We need to prove that the relation holds between each of the combinators, which we do in lemma 3 to 7 and the following remarks. □

Corollary 1 *$\mathcal{P}_{\mathbf{Pgm}\to\mathbf{Id}\to Val^*\to Result}(\mathcal{P}_{std}, \mathcal{P}_\gamma)$ holds, that is, for all programs P, procedure names* I *and pairs of values (v^*, v_γ^*) in $Val_{std}{}^* \times Val_\gamma{}^*$, such that $\mathcal{P}_{Val^*}(v^*, v_\gamma^*)$ holds, we have that $\mathcal{P}_{std}[\![P]\!]\,\mathbf{I}\,v^* \in L(\mathcal{P}_\gamma[\![P]\!]\,\mathbf{I}\,v_\gamma^*)$ holds.*

Proof: Since the logical relations on the domains declared in the core are constructed in the standard way and the core is designed with standard operations, the result follows directly. □

That the initial stores are related is obvious.

The combinators for the primitives can be divided into three kinds: the constructors, the management, and the ones we don't care about.

The ones we don't care about are predicates and destructors such as **car** or **null?**, whose interpretation in γ returns simple results such as **Val** or **Bool**. The proof of safety for this class of primitives relies on the language generating function L taking **Val** to the set of all constants, *etc.*, and is obvious from their definitions.

The constructors, such as **build-primop1**, are only slightly more complicated, relying on the proper handling of alternates as well as L. Following the schema outlined in section 4.2.1, it is obvious that these are also safe.

The only non-obvious constructor is **_sim-build-def**, that not only builds the procedure definition, but also adds a new rule to the grammar. Safety of this scheme follows from property 1.

The combinators for the management operators, such as **gen-proc-name**, and some other, simpler name-generators are not much more complicated. They all take a string argument (and maybe other arguments) and return a pair with the string and a suffix (actually a number), corresponding to a unique string. The post-processor then converts the number into a string and concatenate the two. Thus it is safe to model them in γ as returning $Pair(s, \mathbf{Nat})$, where s is the argument. **gen-proc-name** also returns a boolean value, indicating whether the procedure (and its arguments) has been seen before. This can safely be modeled by returning **Bool**.

That the valuation functions for constants are related is also obvious.

The first lemma concerns the combinator **mk-penv**. Because the grammar-generating interpretation of the domain *Penv* has two components, we need two auxiliary lemmata in the proof of lemma 3. The first of these (lemma 4) establishes that $\mathcal{P}_{Penv}$ holds between the standard environment and the template with any suitable memo-function, not just the initialization. The second (lemma 5) establishes that if sufficient conditions hold for certain elements of a chain, then Q_{CEnv} must hold for smaller elements of the chain.

Lemma 3 *For the combinator* **mk-penv** *we have that*
$\mathcal{P}_{Proc\text{-}Name^*\to Proc\text{-}Template^*\to Penv}(\mathbf{mk\text{-}penv}_{std}, \mathbf{mk\text{-}penv}_\gamma)$
holds.

Proof: We assume we are given $[\mathbf{F}_1,\ldots,\mathbf{F}_n] \in \text{Proc-Name}^*$ and $([\pi_1,\ldots,\pi_n],[\pi_{1\gamma},\ldots,\pi_{n\gamma}])$ in $\mathit{Proc\text{-}Template}^*_{std} \times \mathit{Proc\text{-}Template}^*_{\gamma}$, so that for $j = 1,\ldots,n$ $\mathcal{P}_{\mathit{Proc\text{-}Template}}(\pi_j,\pi_{j\gamma})$ holds.

We clearly have that $\lambda\,\mathbf{F}.\ \mathit{undef}_{CProc}$ fulfills condition Q_{CEnv}.

By lemma 4 below, we then have that $\mathcal{P}_{Penv}$ holds between

$$\mathit{fix}\ \lambda\,\varphi.\,[\mathbf{F}_1 \mapsto \pi_1\,\varphi,\ldots,\mathbf{F}_n \mapsto \pi_n\,\varphi]\lambda\,f.\,\perp_{\mathit{Proc}_{std}}$$

and

$$(\mathit{fix}\ \lambda\,\tau.\,[\mathbf{F}_1 \mapsto \mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_1\,\pi_1\,\tau,\ldots,\mathbf{F}_n \mapsto \mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_n\,\pi_n\,\tau]\lambda\,f.\,\lambda\,\varphi_c.\,\perp_{\mathit{Proc}_{\gamma}},\lambda\,\mathbf{F}.\ \mathit{undef}_{CProc})$$

. □

Abbreviations To improve the readability of the proofs, we will use H to abbreviate

$$\lambda\,\varphi.\,[\mathbf{F}_1 \mapsto \pi_1\,\varphi,\ldots,\mathbf{F}_n \mapsto \pi_n\,\varphi]\lambda\,f.\,\perp_{\mathit{Proc}_{std}}$$

and H_{γ} to abbreviate

$$\lambda\,\tau.\,[\mathbf{F}_1 \mapsto \mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_1\,\pi_{1\gamma}\,\tau,\ldots,\mathbf{F}_n \mapsto \mathit{mk\text{-}proc}_{\gamma}\ F_n\,\pi_{n\gamma}\,\tau]\lambda\,f.\,\lambda\,\varphi_c.\,\perp_{\mathit{Proc}_{\gamma}}$$

whenever we are given $[\mathbf{F}_1,\ldots,\mathbf{F}_n] \in \text{Proc-Name}^*$ and $([\pi_1,\ldots,\pi_n],[\pi_{1\gamma},\ldots,\pi_{n\gamma}])$ in $\mathit{Proc\text{-}Template}^*_{std} \times \mathit{Proc\text{-}Template}^*_{\gamma}$. Furthermore, we will abbreviate $H^i(\perp)$ by φ^i and $H^i_{\gamma}(\perp)$ by τ^i and the least fixed points of the functionals by φ and τ.

Lemma 4 *Let $[\mathbf{F}_1,\ldots,\mathbf{F}_n]$ in Proc-Name* be a list of procedure names and let $([\pi_1,\ldots,\pi_n],[\pi_{1\gamma},\ldots,\pi_{n\gamma}])$ in $\mathit{Proc\text{-}Template}^*_{std} \times \mathit{Proc\text{-}Template}^*_{\gamma}$ be a list of standard and non-standard procedure templates, so that for $j = 1,\ldots,n$ $\mathcal{P}_{\mathit{Proc\text{-}Template}}(\pi_j,\pi_{j\gamma})$ holds.*

Then for all $\varphi_c \in CEnv$ we have that $Q_{CEnv}(\varphi,\varphi_c)$ implies

$$\mathcal{P}_{Penv}(\varphi,(\mathit{fix}\ \lambda\,\tau.\,[\mathbf{F}_1 \mapsto \mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_1\,\pi_1\,\tau,\ldots,\mathbf{F}_n \mapsto \mathit{mk\text{-}proc}_{\gamma}\ F_n\,\pi_n\,\tau]\lambda\,f.\,\lambda\,\varphi_c.\,\perp_{\mathit{Proc}_{\gamma}},\varphi_c))$$

Proof: We use fixed point induction[3]:

Base case: Clearly, for all φ_c in $CEnv$, $Q_{CEnv}(\perp_{\mathit{Penv}_{std}},\varphi_c)$ implies $\mathcal{P}_{Penv}(\perp_{\mathit{Penv}_{std}},(\perp_{\mathit{Template}_{\gamma}},\varphi_c))$.

Step: Assume that for all l,m: $l \le m \le i$ and $\varphi_c \in CEnv$: $Q_{CEnv}(\varphi^l,\varphi_c) \Rightarrow \mathcal{P}_{Penv}(\varphi^l,(\tau^m,\varphi_c))$.

Show that for all $l \le m \le i+1$, $\varphi_c \in CEnv$: $Q_{CEnv}(\varphi^l,\varphi_c) \Rightarrow \mathcal{P}_{Penv}(\varphi^l,(\tau^m,\varphi_c))$.

Assume we are given an l and an m so that $l \le m \le i+1$. If $l = 0$, the result is trivial, so we can assume $l > 0$. Then we can unfold the functionals, to see that we have to show

$Q_{CEnv}(\varphi^l,\varphi_c) \Rightarrow$

$\mathcal{P}_{Penv}([\ldots,\mathbf{F}_j \mapsto \pi_j\,\varphi^{l-1},\ldots]\lambda\,f.\perp_{\mathit{Proc}_{std}},([\ldots,\mathbf{F}_j \mapsto \mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_j\,\pi_{j\gamma}\,\tau^{m-1},\ldots]\lambda\,f.\lambda\,\varphi_c.\perp_{\mathit{Proc}_{\gamma}},\varphi_c))$

So consider a φ_c so that $Q_{CEnv}(\varphi^l,\varphi_c)$.

This assumption means that we only need to prove the right disjoint of $\mathcal{P}_{Penv}$.

We have only one list of names, so any other name will be undefined in both cases. So we assume we are given an $\mathbf{F}_j$ in $[\mathbf{F}_1,\ldots,\mathbf{F}_n]$. Then we have to prove

$\mathcal{P}_{Proc}(\pi_j\,\varphi^{l-1},\mathit{mk\text{-}proc}_{\gamma}\ \mathbf{F}_j\,\pi_{j\gamma}\,\tau^{m-1}\,\varphi_c)$

that is, for all (v^*,v^*_{γ}) in $\mathit{Val}^* \times \mathit{Val}^*_{\gamma}$ and (σ,σ_{γ}) in $\mathit{Store} \times \mathit{Store}_{\gamma}$, so that $\mathcal{P}_{Val^*}(v^*,v^*_{\gamma})$ and $\mathcal{P}_{Store}(\sigma,\sigma_{\gamma})$:

[3] The predicate used here is inductive, since if Q_{CEnv} holds at the limit, it must also hold in the chain.

$$\mathcal{P}_{Result}(\pi_j\ \varphi^{l-1}\ v^*\ \sigma,\ mk\text{-}proc_\gamma\ \mathrm{F}_j\ \pi_{j\gamma}\ \tau^{m-1}\ \varphi_c\ v^*_\gamma\ \sigma_\gamma) \tag{6.1}$$

Now we unfold the definition of *mk-proc* and the generalizations to $(\mathbf{Val}, \gamma_{empty})$ (which we abbreviate v_{Val}), and observe that by property 1 of the languages generated by grammars, 6.1 follows if we can show that

$$\mathcal{P}_{Result}(\pi_j\ \varphi^{l-1}\ v^*\ \sigma, fix\ \lambda\ r_j.\ \pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ r_j)]\varphi_c)\ v_{\mathrm{Val}}{}^*\ (v_{\mathrm{Val}}, v_{\mathrm{Val}}))$$

holds, where α is a sequence of three non-terminals.

Now we unfold the fixed point once, to get $\pi_{j\gamma}$ in the front:

$fix\ \lambda\ r_j.\ \pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ r_j)]\varphi_c)\ v_{\mathrm{Val}}\ (v_{\mathrm{Val}}, v_{\mathrm{Val}})) =$
$\pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res$
α
$(fix\ \lambda\ r_j.\ \pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ r_j)]\varphi_c)$
$v_{\mathrm{Val}}{}^*\ (v_{\mathrm{Val}}, v_{\mathrm{Val}})))]\varphi_c)$
$v_{\mathrm{Val}}{}^*\ (v_{\mathrm{Val}}, v_{\mathrm{Val}})$

Let us abbreviate
$\lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ (fix\ \lambda\ r_j.\ \pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ r_j)]\varphi_c)\ v_{\mathrm{Val}}{}^*\ (v_{\mathrm{Val}}, v_{\mathrm{Val}})))$
by $f^{m-1}_{\gamma j}$.

Now, since we have that π_j is related to $\pi_{j\gamma}$ and the arguments and the store are also related, we only need to show that the respective procedure environments are related, *i.e.*, that

$$\mathcal{P}_{Penv}(\varphi^{l-1}, (\tau^{m-1}, [\mathrm{F}_j \mapsto f^{m-1}_{\gamma j}]\varphi_c))$$

If we can show

$$Q_{CEnv}(\varphi^{l-1}, [\mathrm{F}_j \mapsto f^{m-1}_{\gamma j}]\varphi_c)$$

then it follows from the inductive hypothesis that the procedure environments match, since $l-1 \leq m-1 \leq i$, and we are done. But this follows exactly from lemma 5. □

Lemma 5 *Assume that we have* $[\mathrm{F}_1, \ldots, \mathrm{F}_n]$ *in* Proc-Name* *and* $([\pi_1, \ldots, \pi_n], [\pi_{1\gamma}, \ldots, \pi_{n\gamma}])$ *in* $Proc\text{-}Template^*_{std} \times Proc\text{-}Template^*_\gamma$, *so that for* $j = 1, \ldots, n$ $\mathcal{P}_{Proc\text{-}Template}(\pi_j, \pi_{j\gamma})$ *holds.*

Then, for all $i \in \mathbb{N}$ *and* $j = 1, \ldots, n$, *if we have*

$$\forall l \leq m \leq i, \forall \varphi_c \in CEnv : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow \mathcal{P}_{Penv}(\varphi^l, (\tau^m, \varphi_c))$$

and an l *and an* m *with* $l \leq m \leq i+1$ *and let*
$f^{m-1}_{\gamma j} =$
$\lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ (fix\ \lambda\ r_j.\ \pi_{j\gamma}\ (\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\ v^*\ \sigma.\ (abstract\text{-}res\ \alpha\ r_j)]\varphi_c)\ v_{\mathrm{Val}}{}^*\ (v_{\mathrm{Val}}, v_{\mathrm{Val}})))$
then we have

$$\forall k \leq l : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow Q_{CEnv}(\varphi^k, [\mathrm{F}_j \mapsto f^{m-1}_{\gamma j}]\varphi_c)$$

Proof: Assume we are given an i, so that $\forall l \leq m \leq i, \varphi_c \in CEnv : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow \mathcal{P}_{Penv}(\varphi^l, (\tau^m, \varphi_c))$ and that we are given an l and an m with $l \leq m \leq i+1$.

Then we want to prove that $\forall k \leq l : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow Q_{CEnv}(\varphi^k, [F_j \mapsto f^{m-1}_{\gamma j}]\varphi_c)$. We use induction over k.

k = 0: Clearly $Q_{CEnv}(\lambda\ f.\ \perp_{Proc_{std}}, [\mathrm{F}_j \mapsto f^{m-1}_{\gamma j}]\varphi_c)$ holds in all cases.

Step:

Assume $k < l$ and $\forall h \leq k, \varphi_c \in CEnv : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow Q_{CEnv}(\varphi^h, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)$ holds.
Show $\forall h \leq k+1, \varphi_c \in CEnv : Q_{CEnv}(\varphi^l, \varphi_c) \Rightarrow Q_{CEnv}(\varphi^h, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)$.

So assume that we are given an h with $h \leq k+1$ and a φ_c with $CEnv(\varphi^l, \varphi_c)$.
Since $\varphi^h \sqsubseteq \varphi^l$, it follows that $Q_{CEnv}(\varphi^l, \varphi_c)$ implies $Q_{CEnv}(\varphi^h, \varphi_c)$. So for any $\mathrm{F} \neq \mathrm{F}_j$, we have either $([\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)\,\mathrm{F} = \varphi_c\,\mathrm{F} = undef_{CProc}$ or $\mathcal{P}_{Proc}(\varphi^h \mathrm{F}, ([\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)\,\mathrm{F})$.

For $\mathrm{F} = \mathrm{F}_j$ we have that $\varphi^h\,\mathrm{F}_j\,v^*\,\sigma = \pi_j\,\varphi^{h-1}\,v^*\,\sigma$ so we need to prove for all (v^*, w_γ^*) and (σ, σ'_γ) with $\mathcal{P}_{Var}(v^*, w_\gamma^*)$ and $\mathcal{P}_{Store}(\sigma, \sigma'_\gamma)$ that

$$\mathcal{P}_{Result}(\pi_j\,\varphi^{h-1}\,v^*\,\sigma, f_{\gamma j}^{m-1}\,w_\gamma^*\,\sigma'_\gamma)$$

holds. By using property 1 and unfolding the *fix* in $f_{\gamma j}^{m-1}$ (as before), we get

$f_{\gamma j}^{m-1}\,w_\gamma^*\,\sigma'_\gamma =$
$(abstract\text{-}res\,\alpha\,(fix\ \lambda\,r_j.\ \pi_{j\gamma}\,(\tau^{m-1}, [\mathrm{F}_j \mapsto \lambda\,v^*\,\sigma.\ (abstract\text{-}res\,\alpha\,r_j)]\varphi_c)\,v_{\mathrm{Val}}{}^*\,(v_{\mathrm{Val}}, v_{\mathrm{Val}})))\ \sqsupseteq$
$\pi_{j\gamma}\,(\tau^{m-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)\,v_{\mathrm{Val}}{}^*\,(v_{\mathrm{Val}}, v_{\mathrm{Val}})$
so again we get that $\mathcal{P}_{Result}$ holds if $\mathcal{P}_{Penv}(\varphi^{h-1}, (\tau^{m-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c))$ holds.
Since $h-1 \leq k$ we have that $Q_{CEnv}(\varphi^{h-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)$ holds by the inductive hypothesis.
Since $h-1 \leq m-1 \leq i$ we have

$$Q_{CEnv}(\varphi^{h-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c) \Rightarrow \mathcal{P}_{Penv}(\varphi^{h-1}, (\tau^{m-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c))$$

by assumption. So we conclude that $\mathcal{P}_{Penv}(\varphi^{h-1}, (\tau^{m-1}, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c))$ holds. So we get that $\mathcal{P}_{Result}$ holds as needed, so we have shown $Q_{CEnv}(\varphi^h, [\mathrm{F}_j \mapsto f_{\gamma j}^{m-1}]\varphi_c)$. □

Remarks on lemma 3 to 5.

The inductions in lemma 4 and lemma 5 establish that the appropriate conditions hold when the iterations in the standard and grammar-generating interpretations are "out of step". This again implies that appropriate conditions hold at each step. This is then used to conclude that $\mathcal{P}_{Penv}$ holds at the limit. Along with the trivial fact that $Q_{CEnv}(\varphi, \lambda\,\mathrm{F}.\ undef_{CProc})$ holds for any φ, this is then used to prove lemma 3.

The fixed point in **mk-penv**$_{std}$ is a "real" fixed point in the sense that it denotes the limit of an infinitely increasing chain and cannot be computed in finite time. The fixed points in **mk-penv**$_\gamma$ are both "trivial". The purpose of the inner one is to replace any self-refence in the program (recursive calls) by a self-reference in the result. Since the reference is never looked up (a property of our restricted interpretation of the primitive operations), this can clearly be accomplished with only one unfolding. This is reflected in the proof by not having a fixed point iteration over the inner fixed point, it is sufficient to unfold it. The outer fixed point in **mk-penv**$_\gamma$ becomes trivial because the inner one is, so the recursive references are gradually replaced by references to constants. Otherwise the purpose of the outer fixed point is to mimic the fixed point in the standard interpretation. The gradual replacement by constants ruins the mimicry and is thus responsible for the proof considering "out of step" iterations.

Note that the properties proven will probably not hold in general, only for the elements and limits of the chains constructed from the bottom elements in the semantics.

Lemma 6 *For the combinator* **lookup-proc** *we have that*

$$\mathcal{P}_{Penv \to Proc\text{-}Name \to Proc}(\textbf{lookup-proc}_{std}, \textbf{lookup-proc}_\gamma)$$

holds.

Proof: Assume we are given an F in Proc-Name and $(\varphi,(\tau,\varphi_c))$ in $Penv_{std} \times Penv_\gamma$ so that $\mathcal{P}_{Penv}(\varphi,(\tau,\varphi_c))$.

Then we need to show $\mathcal{P}_{Proc}(\textbf{lookup-proc}_{std}\, F\, \varphi, \textbf{lookup-proc}_\gamma\, F\, (\tau,\varphi_c))$, that is $\mathcal{P}_{Proc}((\varphi\, \mathbf{F}),((\varphi_c\, \mathbf{F}) = \mathit{undef}_{CProc}) \rightarrow (\tau\, F\, \varphi_c) \,[\!]\, \varphi_c\, \mathbf{F})$. But this follows trivially from $\mathcal{P}_{Penv}(\varphi,(\tau,\varphi_c))$. □

Lemma 7 *For the combinator* **cond** *we have that* $\mathcal{P}_{Thunk \rightarrow Thunk \rightarrow Thunk \rightarrow Thunk}(\textbf{cond}_{std}, \textbf{cond}_\gamma)$ *holds.*

Proof: To show this, we assume $\mathcal{P}_{Thunk}(\theta_0,\theta_{\gamma_0})$, $\mathcal{P}_{Thunk}(\theta_1,\theta_{\gamma_1})$, $\mathcal{P}_{Thunk}(\theta_2,\theta_{\gamma_2})$ and $\mathcal{P}_{Store}(\sigma,\sigma_\gamma)$. From this we need to show $\mathcal{P}_{Result}(\textbf{cond}_{std}\,\theta_0\,\theta_1\,\theta_2\,\sigma, \textbf{cond}_\gamma\,\theta_{\gamma_0}\,\theta_{\gamma_1}\,\theta_{\gamma_2}\,\sigma_\gamma)$. If we unfold the definitions, this is

$\mathcal{P}_{Result}(\mathit{let}\ (\mathit{Bool}(b),\sigma_0) = (\theta_0\,\sigma)\ \mathit{in}\ (b \rightarrow \theta_1\,\sigma_0 \,[\!]\, \theta_2\,\sigma_0),$
$\quad \mathit{let}\ (([a_1,\ldots,a_n],\gamma),\sigma_{\gamma_0}) = (\theta_0\,\sigma)$
$\quad \mathit{in}\ ([a_1,\ldots,a_n] = [\mathbf{True}]) \rightarrow (\theta_1\,\sigma_{\gamma_0})$
$\quad\quad [\!]\ ([a_1,\ldots,a_n] = [\mathbf{False}]) \rightarrow (\theta_2\,\sigma_{\gamma_0}) \,[\!]\, \mathit{join}(\theta_1\,\sigma_{\gamma_0})(\theta_2\,\sigma_{\gamma_0}))$

If $(\theta_0\,\sigma) \downarrow 1$ is not a boolean, then the standard result is $\perp_{Result_{std}}$, which is in any language, so the predicate holds.

If $(\theta_0\,\sigma) \downarrow 1$ is $\mathit{Bool}(b)$, then by assumption $b \in L((\theta_{\gamma_0}\,\sigma_\gamma) \downarrow 1)$ and $\mathcal{P}_{Store}(\sigma_0,\sigma_{\gamma_0})$ holds. Now we have three cases:

$(\theta_{\gamma_0}\,\sigma) \downarrow 1 = ([\mathbf{True}],\gamma_{empty})$: since $b \in L((\theta_{\gamma_0}\,\sigma_\gamma) \downarrow 1)$, this implies that $b = \mathit{true}$, so by assumption we are done.

$(\theta_{\gamma_0}\,\sigma) \downarrow 1 = ([\mathbf{False}],\gamma_{empty})$: similar.

$(\theta_{\gamma_0}\,\sigma) \downarrow 1 = ([a_1,\ldots,a_n])$: in this case $(\textbf{cond}_\gamma\,\theta_{\gamma_0}\,\theta_{\gamma_1}\,\theta_{\gamma_2}\,\sigma_\gamma)$ is $\mathit{join}(\theta_{\gamma_1}\,\sigma_0)(\theta_{\gamma_2}\,\sigma_0)$. Then we have either

$\textbf{cond}_{std}\,\theta_0\,\theta_1\,\theta_2\,\sigma = \theta_1\,\sigma_0$ and
$\mathcal{P}_{Result}(\theta_1\,\sigma_0,\theta_{\gamma_1}\,\sigma_{\gamma_0}) \Rightarrow \mathcal{P}_{Result}(\theta_1\,\sigma_0,(\mathit{join}(\theta_{\gamma_1}\,\sigma_{\gamma_0})(\theta_{\gamma_2}\,\sigma_{\gamma_0})))$
or the converse. □

6.2 Termination for the First-Order Grammar-Generating Interpretation

In this section we outline a proof of termination for the grammar-generating semantics and give some loose complexity considerations. We have omitted a formal proof for the sake of brevity.

Complexity could clearly be improved by modifying the algorithm. This is discussed further in chapter 8.

We can use the factorization into a core and an interpretation to subdivide the discussion of termination.

If each of the combinators is terminating, and if the procedures constructed in the combinator $\textbf{mk-penv}_\gamma$ are terminating, then the core with that interpretation is clearly terminating.

The combinators σ_{init_γ}, $\textbf{cond}_\gamma$, and $\textbf{lookup-proc}_\gamma$ are clearly terminating. The simple specifications we have shown for the primitive operations are of course also terminating.

The fixed point in **mk-penv** is terminating if the local fixed points in the procedures are.

Since we do not use a global fixed point, we have now reduced termination of the analysis to the termination of the local fixed points.

Establishing termination of the local fixed points is tricky, because the domain we use is neither finite nor does it have a finite chain property. Instead we argue that the analysis can only construct finite chains in the domain for each given program. An alternate way of obtaining the same result would be to construct a restricted domain, consisting of only the finite chains, and then proving that the analysis is defined on this restricted domain. This would be similar to the approach taken by Mogensen in [Mog88].

We first claim that during computation of the fixed point, the right hand side part of the value will be the same after each iteration.

This is easily seen. The right hand side is the result of analyzing the body of the procedure. We now go through all the possible syntactic forms in the body. If there is a constant in the body, it will clearly return the same value at each iteration. If there is an identifier, it will also return the same value (**Val**), since the arguments are all generalized to **Val**, and the same arguments are passed to the procedure at each iteration. If there is a procedure call, the right hand side of the result will always be the non-terminal for that procedure. If there is a conditional or a let-expression, the result is composed from the results of their subexpressions in a way that is also the same at each iteration. Thus we conclude that the right hand side of the result returned after each iteration of the fixed point must be the same. (This must also hold for the store component of the result.)

This implies that the rule that ***abstract-res*** adds to the grammar in the extension of the memo-function must also be the same after each iteration. In other words, after the first iteration, adding that rule causes no change in the grammar.

Now we only need to establish that the treatment of the body of the procedure does not cause any other kind of change in the grammar after the first iteration. Then we can conclude that the fixed point is trivial and remove it[4].

The only possible changes in the grammar part of the value stem from a procedure call or from the generation of a new residual procedure. In the case of the addition of a residual procedure, the non-terminal and the right hand side of that rule will be the same at each iteration, following the same reasoning as above. In the case of a procedure call, the rule added by ***abstract-res*** will also be the same at each iteration, as a consequence of the reasoning above.

This is not yet quite enough to ensure termination of the procedure application, though, since all of this reasoning is based on the assumption that computation of *one* iteration towards the fixed point will terminate.

That this holds relies on the fact that at each call, the memo-function is extended with the called procedure. So after at most as many calls as there are procedures in the program, any called procedure will be in the memo-function. In other words, the depth of the recursive calls can at most be equal to the number of procedures. And computation of the application of a procedure that is in the memo-function terminates (it is even constant time)

Thus we have termination of the algorithm.

If we want a more precise treatment of the primitives, there is still no problem in showing that **cons** and other constructors are terminating. For the destructors, such as **car**, we have to be a bit more careful if we let them destruct the grammar component of their argument. Since the grammar is designed to capture loops, chasing non-terminals around a grammar to take the car of the actual value might loop if it does not involve a "seen-before" test. Also, our reasoning from above about the fixed-point needs to be revised.

It is clear from the reasoning above, though, that our analysis is not very efficient. We only propagate the memo-function downwards in a call, so if the same procedure is called twice, computation of the local fixed point is repeated.

[4] This is not only part of a termination proof, it also works wonders for our complexity reasoning.

Similarly, at the next iteration of the fixed point, we have to compute all the recursive calls once again. The rule for the called procedure will already be in the grammar (since the result of a deeper nested call is not dependent on the result of an earlier call), but it is not in the memo-function.

So in a worst-case scenario, we would go down a chain of calls as deep as the number of procedures in the program, and at each level, we would compute the fixed point of the called procedure and all its "child" calls and forget about it when returning from the call.

Since we have argued that we can remove the fixed point computation, though, all we actually have to do at each level is to run through the body of the procedure once and treat any calls that are not yet in the memo-function.

This could be simplified further by globalizing the memo-function, which would make the analysis linear in the size of the program, assuming that all primitive operations were constant time.

This is clearly not the case in the current version, though. The scheme for abstract constructors described in section 4.2.1 is clearly not very efficient, but could be replaced by the delayed scheme that is discussed in the same section and that is constant time.

This is discussed further in chapter 8.

6.3 Safety of the Higher-Order Grammar-Generating Interpretation

6.3.1 The core for the higher-order language

The core is given in figure 6.5.

As in the first-order case, the domains of values and procedure environments are unspecified in the core. Since we now have procedures (in the form of closures) in the environment, we also have to leave the specification of environments in the interpretations. There are combinators for extension, lookup, and initialization of environments and procedure environments as well as for closure construction and application of values and for the conditional.

The valuation function $\mathcal{P}$ uses the combinator **mk-init-env** to build the initial environment. The valuation function $\mathcal{L}$ depends only on the program, and so it is entirely defined in the core.

In the valuation function $\mathcal{E}$ on expressions we use the combinator **cond** for handling the conditional and the combinator **apply** for application. For a lambda expression, the combinator **mk-closure** injects a closure into the domain of values.

We leave the valuation function $\mathcal{O}$, for primitive operations, unspecified. Since none of our primitive operations are higher-order, it is essentially equal to the first-order version.

6.3.2 Standard interpretation of the higher-order language

The standard interpretation of the higher-order language is given in figure 6.6.

As before, it is clear that this interpretation of the core in figure 6.5 gives the standard semantics of figure 4.8.

6.3.3 Grammar-generating interpretation of the higher-order language

The grammar-generating interpretation γ of the higher-order language is given in figure 6.7. The interpretation uses the same auxiliary combinators as before. Note that some of the auxiliaries from section 4.3.3 have now been "promoted" to combinators.

Domains:

$v \in \mathit{Val}$ $\quad$ $\Phi \in \mathit{Proc\text{-}Env}$

$c \in \mathit{Closure} = \mathit{Program\text{-}Point} \times \mathit{Env}$ $\quad$ $\sigma \in \mathit{Store} = \mathit{Val} \times \mathit{Val}$

$\rho \in \mathit{Env}$ $\quad$ $(v, \sigma) \in \mathit{Result} = \mathit{Val} \times \mathit{Store}$

$\pi \in \mathit{Proc\text{-}Template} = \mathit{Env} \to \mathit{Proc}$ $\quad$ $\theta \in \mathit{Thunk} = \mathit{Store} \to \mathit{Result}$

$f \in \mathit{Proc} = \mathit{Val}^* \to \mathit{Proc\text{-}Env} \to \mathit{Store} \to \mathit{Result}$ $\quad$ $\varphi \in \mathit{Program\text{-}Point} \to \mathit{Proc\text{-}Template}$

Valuation functions:

$\mathcal{P} : \mathrm{Pgm} \to \mathrm{Id} \to \mathit{Val}^* \to \mathit{Val}$

$\mathcal{P}[\![(\mathtt{define}\ (\mathtt{I}_0\ \mathtt{I}_{1_0}\ \ldots\ \mathtt{I}_{n_0})\ \mathtt{E}_0^{\mathtt{I}_0})\ \ldots\ (\mathtt{define}\ (\mathtt{I}_m\ \mathtt{I}_{1_m}\ \ldots\ \mathtt{I}_{n_m})\ \mathtt{E}_0^{\mathtt{I}_m})]\!]\,\mathtt{I}\,v^* =$

$\quad \mathit{let}\ \varphi_0 = \mathcal{L}[\![\mathtt{E}_0^{\mathtt{I}_0}]\!]([\mathtt{I}_0 \mapsto \lambda\,\rho\,[v_0, \ldots, v_{n_0}]\,\Phi\,\sigma.\ \mathcal{E}[\![\mathtt{E}_0^{\mathtt{I}_0}]\!]\,([\mathtt{I}_{1_0} \mapsto v_1, \ldots, \mathtt{I}_{n_0} \mapsto v_{n_0}]\rho)\,\Phi\,\sigma]$

$\qquad\qquad (\ldots, \mathcal{L}[\![\mathtt{E}_m^{\mathtt{I}_m}]\!]([\mathtt{I}_m \mapsto \lambda\,\rho\,[v_0, \ldots, v_{n_m}]\,\Phi\,\sigma.\ \mathcal{E}[\![\mathtt{E}_m^{\mathtt{I}_m}]\!]$

$\qquad\qquad\qquad ([\mathtt{I}_{1_0} \mapsto v_1, \ldots, \mathtt{I}_{n_m} \mapsto v_{n_m}]\rho)\,\Phi\,\sigma]\,\bot_{\mathit{ProcEnv}})\ldots))$

$\quad \mathit{in}\ \mathit{let}\ \rho_0 = \mathbf{mk\text{-}init\text{-}env}\,[\mathtt{I}_0, \ldots, \mathtt{I}_m]$

$\qquad \mathit{in}\ (\mathbf{apply}\ (\mathbf{lookup\text{-}env}\ \mathtt{I}\,\rho_0)\,v^*\,(\mathbf{init\text{-}proc\text{-}env}\,\varphi_0)\,\sigma_{\mathit{init}})\downarrow 1$

$\mathcal{L} : \mathrm{Pgm} \to (\mathit{Program\text{-}Point} \to \mathit{Proc\text{-}Template}) \to (\mathit{Program\text{-}Point} \to \mathit{Proc\text{-}Template})$

$\mathcal{L}[\![\mathtt{C}]\!]\,\varphi = \varphi$

$\mathcal{L}[\![\mathtt{I}]\!]\,\varphi = \varphi$

$\mathcal{L}[\![(\mathtt{if}\ \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!]\,\varphi = \mathcal{L}[\![\mathtt{E}_0]\!](\mathcal{L}[\![\mathtt{E}_1]\!](\mathcal{L}[\![\mathtt{E}_2]\!]\,\varphi))$

$\mathcal{L}[\![(\mathtt{lambda}\ (\mathtt{I}_1\ \ldots\ \mathtt{I}_n)\ \mathtt{E})^p]\!]\,\varphi =$

$\mathcal{L}[\![\mathtt{E}]\!]\,\varphi([p \mapsto \lambda\,\rho\,[v_1, \ldots, v_n]\,\Phi\,\sigma.\ \mathcal{E}[\![\mathtt{E}]\!]([\mathtt{I}_1 \mapsto v_1, \ldots, \mathtt{I}_n \mapsto v_n]\rho)\,\Phi\,\sigma]\varphi)$

$\ldots$

$\mathcal{E} : \mathrm{Expr} \to \mathit{Env} \to \mathit{Proc\text{-}Env} \to \mathit{Store} \to \mathit{Result}$

$\mathcal{E}[\![\mathtt{C}]\!]\,\rho\,\Phi\,\sigma = (\mathcal{C}[\![\mathtt{C}]\!], \sigma)$

$\mathcal{E}[\![\mathtt{I}]\!]\,\rho\,\Phi\,\sigma = (\mathbf{lookup\text{-}env}\ \mathtt{I}\,\rho, \sigma)$

$\mathcal{E}[\![(\mathtt{if}\ \mathtt{E}_0\ \mathtt{E}_1\ \mathtt{E}_2)]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_0, \sigma_0) = \mathcal{E}[\![\mathtt{E}_0]\!]\,\rho\,\Phi\,\sigma\ \mathit{in}\ \mathbf{cond}\ v_0\,(\mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi)\,(\mathcal{E}[\![\mathtt{E}_2]\!]\,\rho\,\Phi)\,\sigma_0$

$\mathcal{E}[\![(\mathtt{lambda}\ (\mathtt{I}_1\ \ldots\ \mathtt{I}_n)\ \mathtt{E})^p]\!]\,\rho\,\Phi\,\sigma = (\mathbf{mk\text{-}closure}\ (p, \rho), \sigma)$

$\mathcal{E}[\![(\mathtt{O}\ \mathtt{E}_1\ \ldots\ \mathtt{E}_n)]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_1, \sigma_1) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi\,\sigma$

$\qquad\qquad \mathit{in}\ \ldots\ \mathit{let}\ (v_n, \sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\Phi\,\sigma_{n-1}$

$\qquad\qquad\qquad \mathit{in}\ \mathcal{O}[\![\mathtt{O}]\!][v_1, \ldots, v_n]\,\sigma_n$

$\mathcal{E}[\![(\mathtt{E}_0\ \mathtt{E}_1\ \ldots\ \mathtt{E}_n)]\!]\,\rho\,\Phi\,\sigma = \mathit{let}\ (v_0, \sigma_0) = \mathcal{E}[\![\mathtt{E}_1]\!]\,\rho\,\Phi\,\sigma$

$\qquad\qquad \mathit{in}\ \ldots\ \mathit{let}\ (v_n, \sigma_n) = \mathcal{E}[\![\mathtt{E}_n]\!]\,\rho\,\Phi\,\sigma_{n-1}$

$\qquad\qquad\qquad \mathit{in}\ \mathbf{apply}\ v_0[v_1, \ldots, v_n]\,\Phi\,\sigma_n$

where

$\mathbf{init\text{-}proc\text{-}env} : (\mathit{Program\text{-}Point} \to \mathit{Proc}) \to \mathit{Penv}$

$\sigma_{\mathit{init}} : \mathit{Store}$

$\mathbf{mk\text{-}init\text{-}env} : \mathit{Program\text{-}Point}^* \to \mathit{Env}$

$\mathbf{extend\text{-}env} : \mathrm{Id}^* \to \mathit{Val}^* \to \mathit{Env} \to \mathit{Env}$

$\mathbf{lookup\text{-}env} : \mathrm{Id} \to \mathit{Env} \to \mathit{Val}$

$\mathbf{mk\text{-}closure} : \mathit{Closure} \to \mathit{Val}$

$\mathbf{apply} : \mathit{Val} \to \mathit{Val}^* \to \mathit{Proc\text{-}Env} \to \mathit{Store} \to \mathit{Result}$

$\mathbf{cond} : \mathit{Val} \to \mathit{Thunk} \to \mathit{Thunk} \to \mathit{Store} \to \mathit{Result}$

Figure 6.5: Core for the higher-order language

Values:
$v \in Val_{std} = GroundVal_{std} + Closure_{std}$
$g \in GroundVal_{std} = Nat + Bool + Str + Pair_{std} + Res\text{-}Expr$
$(v_1, v_2) \in Pair_{std} = GroundVal \times GroundVal$
Combinators on values:
$\sigma_{init\ std} = (Nil, Nil)$
$\mathbf{mk\text{-}closure}_{std}\,(p, \rho) = in Closure_{std}(p, \rho)$
$\mathbf{apply}_{std}\,v_0\,v^*\,\Phi\,\sigma = let\ Closure(p, \rho_c) = v_0$
$in\ (\Phi\,p)\,\rho_c\,v^*\,\Phi\,\sigma$
$\mathbf{cond}_{std}\,v\,\theta_1\,\theta_2\,\sigma = let\ GroundVal(g) = v_0$
$in\ let\ Bool(b) = g$
$in\ (b \rightarrow (\theta_1\,\sigma_0)\,[\!]\,(\theta_2\,\sigma_0))$

Environments:
$\rho \in Env_{std} = (\mathrm{Id} \rightarrow Val_{std})_\perp$
Combinators on environments:
$\mathbf{mk\text{-}init\text{-}env}_{std}\,[\ldots, p_i, \ldots] = fix\,\lambda\,\rho.\,[\ldots, p_i \mapsto in Closure_{std}(p_i, \rho), \ldots]\lambda\,i.\,\perp_{Val_{std}}$
$\mathbf{extend\text{-}env}_{std}\,[\ldots, \mathrm{I}_i, \ldots]\,[\ldots, v_i, \ldots]\,\rho = [\ldots, \mathrm{I}_i \mapsto v_i, \ldots]\rho$
$\mathbf{lookup\text{-}env}_{std}\,\mathrm{I}\,\rho = \rho\,\mathrm{I}$

Procedure environments:
$\Phi \in Proc\text{-}Env_{std} = Program\text{-}Point \rightarrow Proc\text{-}Template_{std}$
Combinators on procedure environments:
$\mathbf{init\text{-}proc\text{-}env}_{std} = \lambda\,\Phi.\,\Phi$

Figure 6.6: Standard interpretation of the higher-order language

The combinators for environments correspond to the "two-component" environment scheme that we described in chapter 4 and are the same as the auxiliaries given there.

As before, it is clear that this interpretation of the core in figure 6.5 gives the grammar-generating semantics of figure 4.14.

6.3.4 The relation between the higher-order interpretations

The relation we expect to hold between the standard and the grammar-generating interpretations is still that a standard value is in the language generated by the abstract value. So for a standard ground value $GroundVal_{std}(g)$ and an abstract value (c^*, r, γ) consisting of a list of closures, a right hand side and a grammar, we want a relation $\mathcal{R}$ of the form[5]

$$\mathcal{R}_{Val}(GroundVal_{std}(g), (c^*, r, \gamma)) \;=\; g \in L(r, \gamma)$$

to hold between them.

To extend this to closures we can use equality on the program points (the standard closure program point is equal to one of the program points in the list of abstract closures) and the relation

[5]Note that the list c^* of closures in the abstract value is irrelevant. The standard value is a ground value and so cannot be represented as a closure in the abstract value. On the other hand the abstract value is intended to represent a set of standard values, so if it contains closures, this does not detract from the ground values in the set.

Values:

$v_\gamma \in Val_\gamma = Closure_\gamma^* \times Rhs_\gamma \times Grammar_\gamma$

$b \in BaseVal_\gamma = Program\text{-}Point^* \times Rhs_\gamma \times Grammar_\gamma$

$r \in Rhs_\gamma = Alternate_\gamma^*$

$a \in Alternate_\gamma = (Nat^\top + Bool^\top + Str^\top + Pair_\gamma{}^\top + Res\text{-}Expr^\top + Non\text{-}Terminal)^\top$

$(a_1, a_2) \in Pair_\gamma = Alternate_\gamma \times Alternate_\gamma$

$\gamma \in Grammar_\gamma = (Non\text{-}Terminal \times Rhs_\gamma)^*$

Combinators on values:

$\sigma_{init_\gamma} = (([Nil], \gamma_{empty}), ([Nil], \gamma_{empty}))$

$\mathbf{mk\text{-}closure}_\gamma\ (p, \rho) = ([(p, \rho)], [], \gamma_{empty})$

$\mathbf{apply}_\gamma\ ([\ldots, (p_i, \rho_i), \ldots], r, \gamma)\ v^*\ (\mu, \varphi)\ \sigma =$
$(join\text{-}results$
$[\ldots, cases\ (seenbf?\ (p_i, \rho_i, v^*, \sigma)\ \mu)\ of$
$\quad isFound(v, \sigma) \rightarrow (v, \sigma)$
$\quad [] isNotFound() \rightarrow$
$\quad\quad let\ w^* = abstract\text{-}arguments\ p_i\ v^*$
$\quad\quad in\ let\ s' = abstract\text{-}store\ p_i\ \sigma$
$\quad\quad\quad in\ let\ (v_r, \sigma_r) =$
$\quad\quad\quad\quad fix\ \lambda\ (v_r, \sigma_r).\ (\varphi\ p_i)\ \rho_i\ w^*\ ([(p_i, \rho_i, v^*, \sigma) \mapsto abstract\text{-}res\ p_i\ (v_r, \sigma_r)]\mu, \varphi)s'$
$\quad\quad\quad\quad in\ (abstract\text{-}res\ p_i\ (v_r, \sigma_r)),$
$\quad \ldots])$

$\mathbf{cond}_\gamma\ v\ \theta_1\ \theta_2\ \sigma = join\text{-}result\ (\theta_1\ \sigma)\ (\theta_2\ \sigma)$

Environments:

$\rho_\gamma \in Env_\gamma = BaseEnv_\gamma \times EnvEnv_\gamma$

$\beta \in BaseEnv_\gamma = (\mathrm{Id} \times BaseVal_\gamma)^*$

$\kappa \in EnvEnv_\gamma = (Program\text{-}Point \times BaseEnv_\gamma)^*$

Combinators on environments:

$\mathbf{mk\text{-}init\text{-}env}_\gamma\ [\ldots, p_i, \ldots] = let\ \beta = [\ldots, (p_i, ([p_i], [], \gamma_{empty})), \ldots]$
$\quad in\ (\beta, [\ldots, (p_i, \beta), \ldots])$

$\mathbf{extend\text{-}env}_\gamma\ [\ldots, \mathrm{I}_i, \ldots]\ [\ldots, ([\ldots, (p_{ij}, (\beta_{ij}, \kappa_{ij})), \ldots], r_i, \gamma_i), \ldots]\ (\beta, \kappa) =$
$\quad let\ \kappa' = join\text{-}\kappa\ [\ldots, \kappa_{ij}, \ldots, \kappa]$
$\quad in\ let\ \kappa'' = extend\text{-}env\text{-}env\ [\ldots, (p_{ij}, \beta_{ij}), \ldots]\ \kappa'$
$\quad\quad in\ ([\ldots, \mathrm{I}_i \mapsto ([\ldots, p_{ij}, \ldots], r_i, \gamma_i), \ldots]\beta, \kappa'')$

$\mathbf{lookup\text{-}env}_\gamma\ \mathrm{I}\ (\beta, \kappa) = let\ ([\ldots, p_i, \ldots], r, \gamma) = lookup\ \mathrm{I}\ \beta$
$\quad in\ ([\ldots, (p_i, ((lookup\ p_i\ \kappa), \kappa)), \ldots], r, \gamma)$

Procedure environments:

$\Phi, (\mu, \varphi) \in Proc\text{-}Env_\gamma = Memo \times (Program\text{-}Point \rightarrow Proc\text{-}Template_\gamma)$

$\mu \in Memo = ((Program\text{-}Point \times Env_\gamma \times Val_\gamma \times Store_\gamma) \times (Val_\gamma \times Store_\gamma))^*$

Combinators on procedure environments:

$\mathbf{init\text{-}proc\text{-}env}_\gamma = \lambda\ \varphi.\ (\mu_{init}, \varphi)$

Figure 6.7: Grammar-generating interpretation of the higher-order language

on environments for the corresponding environments. So we want $\mathcal{R}$ on closures to look something like this:

$$\mathcal{R}_{Val}(Closure_{std}(p,\rho),([\ldots,(p_j,\rho_j),\ldots])) \;=\; \exists j : p = p_j \wedge \mathcal{R}_{Env}(\rho,\rho_j)$$

Since the structure of the environments are defined in the interpretations, we also need to specify the relation on environments explicitly. We choose the most obvious: for any I in Id, the relation on values holds between $(\textbf{lookup-env}_{std}\,\mathrm{I}\,\rho_{std})$ and $(\textbf{lookup-env}_{\gamma}\,\mathrm{I}\,\rho_{\gamma})$. So the relation on environments should be something like:

$$\mathcal{R}_{Env}(\rho,\rho_\gamma) \;=\; \forall \mathrm{I} : \mathcal{R}_{Val}(\textbf{lookup-env}_{std}\,\mathrm{I}\,\rho, \textbf{lookup-env}_{\gamma}\,\mathrm{I}\,\rho_\gamma)$$

Thus the relation on environments is defined in terms of the relation on values and the relation on (closure) values in terms of the relation on environments. To resolve the circularity, we use a countable family of pairs of relations on values and environments. The first pair of relations is defined only on ground (standard) values and (standard) environments binding identifiers to ground values. The second pair is defined on ground values and closures with environments containing ground values and on environments containing ground values and closures with environments containing only ground values, and so on.

We define the family as follows:

$$\begin{aligned}
\mathcal{R}^0_{Val}(v,(c^*,r,\gamma)) &= (v = GroundVal_{std}(g)) \wedge g \in L(r,\gamma)\\
\mathcal{R}^{i+1}_{Val}(GroundVal_{std}(g),(c^*,r,\gamma)) &= g \in L(r,\gamma)\\
\mathcal{R}^{i+1}_{Val}(Closure_{std}(p,\rho),([\ldots,(p_j,(\beta_j,\kappa_j)),\ldots]^{j=1..n},r,\gamma)) &= \exists k \in 1..n : p = p_k \wedge \mathcal{R}^i_{Env}(\rho,(\beta_k,\kappa_k))
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}^i_{Env}(\rho,(\beta,\kappa)) \;=\; & \forall \mathrm{I} : \mathit{let}\ ([\ldots,p_j,\ldots],r,\gamma) = (\mathit{lookup}\ \mathrm{I}\ \beta)\\
& \quad \mathit{in}\ \mathcal{R}^i_{Val}((\rho\,\mathrm{I}),([\ldots,(p_j,(\mathit{lookup}\ p_j\ \kappa,\kappa)),\ldots],r,\gamma))
\end{aligned}$$

If $\mathcal{R}^i_{Val}(v,v_\gamma)$ holds, we clearly have that $\mathcal{R}^{i+1}_{Val}(v,v_\gamma)$ holds, so $(\mathcal{R}^i_{Val})$ is a chain in $Val_{std} \times Val_\gamma \to \mathbf{2}$. We define the relation $\mathcal{R}_{Val}$ as the limit of the $\mathcal{R}^i_{Val}$ and the relation $\mathcal{R}_{Env}$ as the limit of the $\mathcal{R}^i_{Env}$.

Finally we need to specify the relation on procedure environments. As in the first-order case, the abstract procedure environments have two components while the standard interpretation has only one. We define the relation on procedure environments by relating both the abstract components to the standard procedure environment. But here, the second abstract component is the actual procedure environment, making the relation simpler: it is just the induced relation on the type of procedure environments. The first component is the memo function, which only contain entries for those procedures that have already been treated. So for this component (as before) we let the relation hold if there is no entry for a procedure or if there is an entry and it matches the concrete entry.

$$\begin{aligned}
\mathcal{R}_{Proc\text{-}Env}(\Phi,(\mu,\varphi_\gamma)) = \; & \mathcal{R}_{Program\text{-}Point \to Proc\text{-}Template}(\Phi,\varphi_\gamma) \wedge\\
& \forall p \in Program\text{-}Point, (\rho,\rho_\gamma) \in Env_{std} \times Env_\gamma,\\
& (v^*,v^*_\gamma) \in Val_{std} \times Val_\gamma, (s,s_\gamma) \in Store_{std} \times Store_\gamma :\\
& \mathcal{R}_{Env}(\rho,\rho_\gamma) \wedge \mathcal{R}_{Val}(v^*,v^*_\gamma) \wedge \mathcal{R}_{Store}(s,s_\gamma) \Rightarrow\\
& (seenb4?\,(p,\rho_\gamma,v^*_\gamma,s_\gamma)\,\mu) = NotFound() \vee\\
& ((seenb4?\,(p,\rho_\gamma,v^*_\gamma,s_\gamma)\,\mu) = Found(v',s') \wedge \mathcal{R}_{Result}(\Phi\,p\,\rho\,v^*\,\Phi\,s,(v',s')))
\end{aligned}$$

From these definitions on the base domains of the core, the relation is extended in the usual way to products, sums, and function domains [Nie89].

6.3.5 Proof of safety

In order to prove the safety of the grammar-generating semantics, we prove that the grammar-generating interpretation γ is safe with respect to the standard interpretation *std*. This falls in two parts, the combinators abstracting the primitive operations and the combinators responsible for our abstract handling of environments, conditionals and procedures, that we have defined in figure 6.7.

Theorem 3 *The relation $\mathcal{R}$ holds between std and γ.*

Proof: We need to prove that the relation holds between each of the combinators, which we do in lemma 8 to 14 and the following remarks. □

Corollary 2 *$\mathcal{R}_{\mathbf{Pgm}\to\mathbf{Id}\to Val^* \to Val}(\mathcal{P}_{std}, \mathcal{P}_\gamma)$ holds, that is, for all programs P, procedure names* I *and pairs of values (v^*, v_γ^*) in $Val_{std}^* \times Val_\gamma^*$, such that $\mathcal{R}_{Val^*}(v^*, v_\gamma^*)$ holds, we have that $\mathcal{R}_{Val}(\mathcal{P}_{std}[\![P]\!]\,\mathbf{I}\,v^*, \mathcal{P}_\gamma[\![P]\!]\,\mathbf{I}\,v_\gamma^*)$ holds.*

Proof: Since the logical relations on the domains declared in the core are constructed in the standard way and the core is designed with standard operations, the result follows directly. □

The primitives are essentially the same as in the first-order interpretation, and it is clear that safety in the first-order case carries over (since the higher-order abstract domain of values is an extension of the first-order).[6]

Since, in our higher-order factorization, we have moved the management of the memo-function from the procedure itself to a separate **apply** combinator, the most complex part of the proof of safety is now the lemma for **apply**, rather than the initialization of the procedure environment.

It is obvious that $\mathcal{R}_{Store}$ holds between the two initial stores.

Lemma 8 *$\mathcal{R}_{(Program\text{-}Point\to Proc\text{-}Template)^* \to Proc\text{-}Env}(\mathbf{init\text{-}proc\text{-}env}_{std}, \mathbf{init\text{-}proc\text{-}env}_\gamma)$ holds, that is, for all $\Phi \in Program\text{-}Point \to Proc_{std}$ and $\varphi_\gamma \in Program\text{-}Point \to Proc_\gamma$ so that $\mathcal{R}_{Program\text{-}Point\to Proc}(\Phi, \varphi_\gamma)$ holds, we have that $\mathcal{R}_{Proc\text{-}Env}(\mathbf{init\text{-}proc\text{-}env}_{std}\,\Phi, \mathbf{init\text{-}proc\text{-}env}_\gamma\,\varphi_\gamma)$ holds.*

Proof: The first half is trivial and, since μ_{init} is empty, it is clear that the second part of $\mathcal{R}_{Proc\text{-}Env}$ is also true. □

Lemma 9 *$\mathcal{R}_{Program\text{-}Point^* \to Env}(\mathbf{mk\text{-}init\text{-}env}_{std}, \mathbf{mk\text{-}init\text{-}env}_\gamma)$ holds, that is, for all $[\ldots, p_i, \ldots] \in Program\text{-}Point^*$ we have that $\mathcal{R}_{Env}(\mathbf{mk\text{-}init\text{-}env}_{std}[\ldots, p_i, \ldots], \mathbf{mk\text{-}init\text{-}env}_\gamma[\ldots, p_j, \ldots])$ holds.*

Proof: We assume that we are given $[\ldots, p_i, \ldots] \in Program\text{-}Point^*$. We need to show that for all identifiers $\mathbf{I} \in \mathbf{Id}$:
$\mathcal{R}_{Val}(\mathbf{lookup\text{-}env}_{std}\,\mathbf{I}\,(\mathbf{mk\text{-}init\text{-}env}_{std}[\ldots, p_i, \ldots]), \mathbf{lookup\text{-}env}_\gamma\,\mathbf{I}\,(\mathbf{mk\text{-}init\text{-}env}_\gamma[\ldots, p_j, \ldots]))$
holds.

We first observe that the identifier I will be bound in the standard environment exactly when it is bound in the non-standard environment. So if I is not the name of one of the procedures being bound, the result of looking it up is an error in both cases.

[6] There are obviously also other safe abstractions of the primitives with better precision.

Since **mk-init-env**$_{std}$ is defined with a fixed point, we proceed by fixed point induction over the first component of the relation. For readability we abbreviate the functional $\lambda\ \rho.\ [\ldots, p_i \mapsto in Closure_{std}(p_i, \rho), \ldots]\lambda\ i.\ \perp_{Val_{std}}$ by H. (Note that we actually have a different H for each set of arguments to the combinators.)

- Base case: Prove that $\mathcal{R}_{Env}(\perp_{Env_{std}}, (\textbf{mk-init-env}_\gamma\ [\ldots, p_j, \ldots]))$ holds. This is obvious.
- Induction step: Assume $\mathcal{R}_{Env}(\rho, (\textbf{mk-init-env}_\gamma\ [\ldots, p_j, \ldots]))$ holds. Prove that $\mathcal{R}_{Env}(H(\rho), (\textbf{mk-init-env}_\gamma\ [\ldots, p_j, \ldots]))$ holds.

  It is sufficient to consider the identifiers that are bound in the environments. Assume that the identifier I is equal to p_k for some k. We then have that

  $$\textbf{lookup-env}_{std}\ \mathrm{I}\ ([\ldots, p_i \mapsto in Closure_{std}(p_i, \rho), \ldots]\lambda\ i.\ \perp_{Val_{std}}) = Closure_{std}(p_k, \rho)$$

  and that

  $$\textbf{lookup-env}_\gamma\ \mathrm{I}\ (\textbf{mk-init-env}_\gamma\ [\ldots, p_j, \ldots]) = ([(p_k, (\textbf{mk-init-env}_\gamma\ [\ldots, p_j, \ldots]))], [], \gamma_{empty})$$

  so clearly $\mathcal{R}_{Val}$ holds. □

Lemma 10 $\mathcal{R}_{\mathrm{Id}\to Env\to Val}(\textbf{lookup-env}_{std}, \textbf{lookup-env}_\gamma)$ *holds, that is, for all* $\mathrm{I} \in \mathrm{Id}$, $\rho \in Env_{std}$, *and* $\rho_\gamma \in Env_\gamma$ *such that* $\mathcal{R}_{Env}(\rho, \rho_\gamma)$ *holds, we have that* $\mathcal{R}_{Val}(\textbf{lookup-env}_{std}\ \mathrm{I}\ \rho, \textbf{lookup-env}_\gamma\ \mathrm{I}\ \rho_\gamma)$ *holds.*

Proof: Trivial from the definition of $\mathcal{R}_{Env}$. □

Lemma 11 $\mathcal{R}_{\mathrm{Id}^*\to Val^*\to Env\to Env}(\textbf{extend-env}_{std}, \textbf{extend-env}_\gamma)$ *holds, that is, for all* $\mathrm{I} \in \mathrm{Id}$, $v \in Val_{std}$, $v_\gamma \in Val_\gamma$, $\rho \in Env_{std}$, *and* $\rho_\gamma \in Env_\gamma$, *such that* $\mathcal{R}_{Val}(v, v_\gamma)$ *and* $\mathcal{R}_{Env}(\rho, \rho_\gamma)$ *hold, we have that* $\mathcal{R}_{Env}(\textbf{extend-env}_{std}\ \mathrm{I}\ v\ \rho, \textbf{extend-env}_\gamma\ \mathrm{I}\ v_\gamma\ \rho_\gamma)$ *holds.*

Proof: Since $\mathcal{R}_{Env}$ is defined using the lookup functions, we need to establish that all bindings in the extended environments match. If an identifier J is bound to a ground value in the standard environment, it follows from the prerequisites that it will match the binding of J in the abstract environment. If J is bound to a closure, however, we rely on the fact that **extend-env**$_\gamma$ takes an upper bound of the existing closure environments and the closure environments of v_γ. So for all $\mathrm{J} \neq \mathrm{I}$, we have that $\textbf{lookup-env}_\gamma\ \mathrm{J}\ \rho_\gamma \sqsubseteq_{Val} \textbf{lookup-env}_\gamma\ \mathrm{J}\ (\textbf{extend-env}_\gamma\ \mathrm{I}\ v_\gamma\ \rho_\gamma)$ and similarly $v_\gamma \sqsubseteq_{Val} \textbf{lookup-env}_\gamma\ \mathrm{I}\ (\textbf{extend-env}_\gamma\ \mathrm{I}\ v_\gamma\ \rho_\gamma)$. From this it clearly follows that $\mathcal{R}_{Val}$ holds. □

Lemma 12 $\mathcal{R}_{Val\to Val^*\to Proc\text{-}Env\to Store\to Result}(\textbf{apply}_{std}, \textbf{apply}_\gamma)$ *holds, that is, for all* $v \in Val_{std}$, $v_\gamma \in Val_\gamma$, $v^* \in Val_{std}^*$, $v_\gamma^* \in Val_\gamma^*$, $\Phi_{std} \in Proc\text{-}Env_{std}$, $(\mu, \varphi_\gamma) \in Proc\text{-}Env_\gamma$, $\sigma \in Store_{std}$, *and* $\sigma_\gamma \in Store_\gamma$ *such that* $\mathcal{R}_{Val}(v, v_\gamma)$, $\mathcal{R}_{Val^*}(v^*, v_\gamma^*)$, $\mathcal{R}_{Proc\text{-}Env}(\Phi_{std}, (\mu, \varphi_\gamma))$, *and* $\mathcal{R}_{Store}(\sigma, \sigma_\gamma)$ *hold, we have that* $\mathcal{R}_{Result}(\textbf{apply}_{std}\ v\ v^*\ \Phi_{std}\sigma, \textbf{apply}_\gamma\ v_\gamma\ v_\gamma^*\ (\mu, \varphi_\gamma)\sigma_\gamma)$ *holds.*

Proof: As in the first-order case, we first observe that we can ignore the introduction of non-terminals, since the language generated by the grammar before the introduction will be a subset of the language generated by the grammar after the introduction (property 1).

Now assume that we are given $v \in Val_{std}$, $v_\gamma \in Val_\gamma$, $v^* \in Val^*_{std}$, $v^*_\gamma \in Val^*_\gamma$, $\Phi_{std} \in Proc\text{-}Env_{std}$, $(\mu, \varphi_\gamma) \in Proc\text{-}Env_\gamma$, $\sigma_{std} \in Store_{std}$, and $\sigma_\gamma \in Store_\gamma$ such that $\mathcal{R}_{Val}(v, v_\gamma)$, $\mathcal{R}_{Val^*}(v^*, v^*_\gamma)$, $\mathcal{R}_{Proc\text{-}Env}(\Phi_{std}, (\mu, \varphi_\gamma))$, and $\mathcal{R}_{Store}(\sigma_{std}, \sigma_\gamma)$ hold.

We can assume that $v = Closure_{std}(p, \rho)$ and that $v_\gamma = ([\ldots, (p_i, \rho_{\gamma i}), \ldots], r, \gamma)$. Let k be the index such that $(p_k, \rho_{\gamma k})$ matches v.

Since $\mathbf{apply}_\gamma$ takes an upper bound of the application of all the p_i, we can ignore all but $(p_k, \rho_{\gamma k})$. Consider $(seenb4?\ (p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma)\,\mu)$. There are two possibilities:

- $(seenb4?\ (p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma)\,\mu) = Found(v, \sigma)$: in this case we are done, by our assumptions on Φ_{std} and μ.
- $(seenb4?\ (p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma)\,\mu) = NotFound()$: in this case we need to use fixed point induction over the abstract component. To initialize this, we first introduce a redundant fixed point in the standard combinator: we observe that we can replace $(\Phi_{std}\, p)$ by $(fix\ \lambda\ f.\ \lambda\ \rho\, v^*\, \Phi\, \sigma.\ \Phi_{std}\, p\, \rho\, v^*\, ([p \mapsto f]\Phi_{std})\, \sigma)$.

  Using this definition, we prove that for all $\rho \in Env_{std}$, $w^* \in Val^*_{std}$, $\sigma' \in Store_{std}$ so that $\mathcal{R}_{Env}(\rho, \rho_{\gamma k})$, $\mathcal{R}_{Val^*}(w^*, v^*_\gamma)$, and $\mathcal{R}_{Store}(\sigma', \sigma_\gamma)$ hold, we have

  $$\mathcal{R}_{Result}((fix\ \lambda\ f.\ \lambda\ \rho\, v^*\, \Phi\, \sigma.\ \Phi_{std}\, p\, \rho\, v^*\, ([p \mapsto f]\Phi_{std})\, \sigma)\, \rho\, w^*\, \Phi_{std}\, \sigma',\ fix\ \lambda\ (v_r, \sigma_r).\ \varphi_\gamma\, p_k\, \rho_{\gamma k}\, v^*_\gamma\, ([(p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma) \mapsto (v_r, \sigma_r)]\mu, \varphi_\gamma)\, \sigma_\gamma).$$

  - Base case: obvious.
  - Step: Let us abbreviate the i'th iteration of the fixed-point over standard procedure templates by f^i and the i'th iteration of the fixed-point over the abstract result by (v^i_r, σ^i_r).

    Now assume that at iteration i we have for all $\rho \in Env_{std}$, $w^* \in Val^*_{std}$ and $\sigma' \in Store_{std}$, if we have $\mathcal{R}_{Env}(\rho, \rho_{\gamma k})$, $\mathcal{R}_{Val^*}(w^*, v^*_\gamma)$, and $\mathcal{R}_{Store}(\sigma', \sigma_\gamma)$, then $\mathcal{R}_{Result}(f^i\, \rho\, w^*\, \Phi_{std}\, \sigma', (v^i_r, \sigma^i_r))$ holds.

    From this we have to prove that at iteration $i+1$ we have for all $\rho \in Env_{std}$, $w^* \in Val^*_{std}$ and $\sigma' \in Store_{std}$, if $\mathcal{R}_{Env}(\rho, \rho_{\gamma k})$, $\mathcal{R}_{Val^*}(w^*, v^*_\gamma)$, and $\mathcal{R}_{Store}(\sigma', \sigma_\gamma)$, then $\mathcal{R}_{Result}(f^{i+1}\, \rho\, w^*\, \Phi_{std}\, \sigma', (v^{i+1}_r, \sigma^{i+1}_r))$.

    So we unfold both abbreviations, giving us

    $$\mathcal{R}_{Result}(\Phi_{std}\, p\, \rho\, w^*\, ([p \mapsto f^i]\Phi_{std})\, \sigma', \varphi_\gamma\, p_k\, \rho_{\gamma k}\, v^*_\gamma\, ([(p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma) \mapsto (v^i_r, \sigma^i_r)]\mu, \varphi_\gamma)\, \sigma_\gamma)$$

    that we need to prove.

    Since we have $\mathcal{R}_{Program\text{-}Point \to Proc\text{-}Template}(\Phi_{std}, \varphi_\gamma)$ and for the arguments to the procedure environments we already have that $p = p_k$, $\mathcal{R}_{Env}(\rho, \rho_{\gamma k})$, $\mathcal{R}_{Val^*}(w^*, v^*_\gamma)$, and $\mathcal{R}_{Store}(\sigma', \sigma_\gamma)$, we only need to show

    $$\mathcal{R}_{Proc\text{-}Env}([p \mapsto f^i]\Phi_{std}, ([(p_k, \rho_{\gamma k}, v^*_\gamma, \sigma_\gamma) \mapsto (v^i_r, \sigma^i_r)]\mu, \varphi_\gamma))$$

    For the second component in the abstract procedure environment, this is trivial, since $[p \mapsto f^i]\Phi_{std}$ must be smaller than Φ_{std}, which matches φ_γ.

For the memo-function all we need to show is that the extension of the memo-function matches f^i, since match for the rest of the entries follows from the assumption and the fact that f^i is smaller than $\Phi_{std}\, p$.

So we need to show that

$\forall \rho \in Env_{std}, w^* \in Val^*_{std}, s \in Store_{std} : \mathcal{R}_{Env}(\rho, \rho_\gamma k) \wedge \mathcal{R}_{Val^*}(w^*, v^*_\gamma) \wedge \mathcal{R}_{Store}(\sigma', \sigma_\gamma) \Rightarrow \mathcal{R}_{Result}(f^i\, \rho\, w^*\, ([p \mapsto f^i]\Phi_{std})\, \sigma', (v^i_r, \sigma^i_r))$

Since $[p \mapsto f^i]\Phi_{std}$ is smaller than Φ_{std} this follows from the inductive hypothesis. □

Lemma 13 $\mathcal{R}_{Closure\_Val}(\mathbf{mk\text{-}closure}_{std}, \mathbf{mk\text{-}closure}_\gamma)$ *holds, that is, for all* $(p, \rho) \in Closure_{std}$ *and* $(p', \rho_\gamma) \in Closure_\gamma$ *such that* $\mathcal{R}_{Closure}((p, \rho), (p', \rho_\gamma))$ *holds, we have that* $\mathcal{R}_{Val}(\mathbf{mk\text{-}closure}_{std}\,(p, \rho), \mathbf{mk\text{-}closure}_\gamma\,(p', \rho_\gamma))$ *holds.*

Proof: Obvious.

Lemma 14 $\mathcal{R}_{Val\to Thunk\to Thunk\to Store\to Result}(\mathbf{cond}_{std}, \mathbf{cond}_\gamma)$ *holds, that is, for all* $v \in Val_{std}$, $v_\gamma \in Val_\gamma$, $\theta_1 \in Thunk_{std}$, $\theta_{\gamma 1} \in Thunk_\gamma$, $\theta_2 \in Thunk_{std}$, $\theta_{\gamma 2} \in Thunk_\gamma$, $\sigma \in Store_{std}$, *and* $\sigma_\gamma \in Store_\gamma$, *such that* $\mathcal{R}_{Val}(v, v_\gamma)$, $\mathcal{R}_{Thunk}(\theta_1, \theta_{\gamma 1})$, $\mathcal{R}_{Thunk}(\theta_2, \theta_{\gamma 2})$ *and* $\mathcal{R}_{Store}(\sigma, \sigma_\gamma)$ *hold, we have that* $\mathcal{R}_{Result}(\mathbf{cond}_{std}\, v\, \theta_1\, \theta_2\, \sigma, \mathbf{cond}_\gamma\, v_\gamma\, \theta_{\gamma 1}\, \theta_{\gamma 2}\, \sigma_\gamma)$ *holds.*

Proof: As usual for the conditional, this relies on the fact that we take an upper bound of the two possible results in the abstract interpretation. This is similar to the proof for the first-order conditional. □

6.4 Termination for the Higher-Order Grammar-Generating Interpretation

Since the application of procedures has been moved to a combinator in the higher-order case, we have that if all combinators in an interpretation are terminating, the higher-order core with that interpretation will also terminate.

Termination is straight-forward for all combinators in the higher-order grammar-generating interpretation except $\mathbf{apply}_\gamma$.

Termination of $\mathbf{apply}_\gamma$ relies on the termination of treatment for each of the closures in the closure-list, that is, that the procedures produced by *mk-proc* terminate.

This again relies on the memo-function, as in the first-order case. But we need to argue a little harder to see that all calls can eventually be treated by looking up in the memo-function.

As before, to show termination of the treatment of a procedure, we need to show that the analysis will only construct finitely many values for any given program. This is harder in the higher-order case, for three reasons:

- We construct non-terminals for the parameters as well, so we cannot rely on the generalization to **Val** to conclude that an identifier will always return the same value. On the other hand, since we exactly introduce a non-terminal for the parameter, we have that an identifier will always give that non-terminal as right hand side component when it is looked up. So this is still constant. So we still have that the right hand side component of a value will be the same after each iteration towards the fixed point.

- We pass higher-order values as arguments to procedures, and we do not generate non-terminals for these. On the other hand, we have that if there are only finitely many ground values, there will only be finitely many closures.
- When we look up in the memo-function, we also distinguish between arguments to find an entry. On the other hand, if there are only finitely many different arguments, all calls will still eventually end up in the memo-function.

So we still have that any constructor in a procedure can only operate on syntactic constants and non-terminals (corresponding to calls or to the formal parameters of the procedure) and on right hand sides constructed in the *same* invocation of the procedure. This gives a syntactic limit on the depth of constructions in a right hand side.

We already know that the set of non-terminals is finite for a given program. So for any given program, we can only construct finitely many different right hand sides, in spite of the domain being infinite. This also means that there can be only finitely many grammars.

By the way we have constructed closure environments, this again implies that there can be only finitely many closures, so there are only finitely many values constructed during the analysis of a given program.

We properly extend the memo-function at each call, and the extension is only forgotten *after* we return from the call. So eventually, all calls that we encounter will be in the memo-function. Calls are always looked up in the memo-function, and if they are found, the analysis certainly terminates. So each iteration of the fixed point computation will terminate.

By finiteness of the values, we also have that a fixed point will be reached, so we have that analysis of procedure application will terminates.

Chapter 7

Implementation of the First-Order Analysis

We have implemented a prototype of the first-order analysis in Scheme. The purpose of the implementation is to study the quality of the results of the analysis and we have not given much thought to efficiency.

We aim for an implementation that follows the semantics as closely as possible, essentially transliterating the semantics equations into Scheme. Since the semantics uses mainly strict functions, the call-by-value semantics of Scheme does not cause problems. Since the bottom element in the abstract domain of values is just the empty right hand side and the empty grammar, implementing a non-strict function over abstract values (such as *abstract-res*) does not cause any problems. The fixed point over the template is implemented with `letrec` and a lambda abstraction. The fixed point over the result in application is implemented with a simple loop (since it is finite).

The main work in transliterating the analysis is to design a representation of abstract values and procedure environments. Similix already delivers the generating extension as abstract syntax, so we do not need a parser, we can simply disable the un-parser of Similix. And the actual algorithm is a straightforward transliteration.

We use a simple representation of right hand sides and grammars as lists. Conceptually, they are sets, however, and we need to have join and equality operations on them. For this reason we order them, according to an arbitrary ordering that we impose. This way we can avoid duplicates and still take the union of two sets or compare their elements relatively quickly.

In figure 7.1, we show the main procedures of our implementation, corresponding to the valuation functions. We do not show the entire definition of `_o`, since the cases in this procedure are so similar. All the procedures are essentially identical to the semantics, except that they are uncurried. The main syntactic difference is that the semantic equations are written as if we use a pattern matching dispatch on the syntax, whereas in the Scheme procedures we use separate predicates and destructors.

The procedure `_p` uses the auxiliary `mk-ptemplate` for some of this de-structuring. The procedure `_e` uses an auxiliary `_es` to handle a list of expressions returning the list of values.

In figure 7.2 we show the implementation of some of the combinators. As for the valuation functions, these are transliterated directly from the semantic definitions, except that we cannot define the procedure environment with a fixed point in Scheme. Instead, we define the higher-order recursive procedure environment with a letrec-expression[1].

In figure 7.3 we give the implementation of some of the auxiliaries. These also follow the semantic

[1] This is a particularly inefficient approach, but in the spirit of the semantics.

```
; Pgm x Id x Val* -> Val
(define (_p pgm fname vals)
    (let ((names (get-names-pgm pgm))
          (ptemplates (map mk-ptemplate (get-defs-pgm pgm))))
        (let ((phi (mk-penv names ptemplates)))
            (get-val-res ((lookup-proc fname phi) vals sigma-init)))))

; Expr x Env x Penv -> Store -> Result
(define (_e expr rho phi)
    (lambda (sigma)
        (record-case expr
         (cst (c) (mk-res (mk-val (mk-rhs (convert-constant-to-alternate c)) empty-grammar)
                          sigma))
         (var (i) (mk-res (lookup i rho) sigma))
         (cond (e1 e2 e3)  (_cond (_e e1 rho phi) (_e e2 rho phi) (_e e3 rho phi) sigma))
         (primop (p . es) (_o p (_es es rho phi sigma)))
         (pcall (f es) (let ((res (_es es rho phi sigma)))
                         (let ((vals (get-vals-ress res))
                               (sigma (get-store-ress res)))
                           ((lookup-proc f vals phi) vals sigma))))
         (let (i t e1 e2) (let ((result ((_e e1 rho phi) sigma)))
                            (let ((val (get-val-res result))
                                  (sigma (get-store-res result)))
                              ((_e e2 (extend-env (list i) (list val) rho) phi)
                               sigma))))
         (else (error '_e "unknown syntactic form: ~s" expr)))))

; Op x (Val* x Store) x Id -> Result
(define (_o op result-list)
    (let ((vals (get-vals-ress result-list))
          (sigma (get-store-ress result-list)))
        (case op ('car (mk-res (apply-un-op-val car-rhs (car vals)) sigma))
                 ('cons (mk-res (apply-bin-op-vals cons-rhss (car vals) (cadr vals)) sigma))
          ...
          ('_sim-build-primop1 (mk-res (apply-bin-op-vals build-primop1-rhss
                                                          (car vals) (cadr vals))
                                       sigma))
          ('_sim-generate-proc-name!
           (let ((n (car vals))
                 (p* (cadr vals))
                 (s1 (get-store1-store sigma)))
             (abs-generate-proc-name! (get-rhs-val n) (get-grammar-val n)
                                      (get-rhs-val p*) (get-grammar-val p*)
                                      (get-rhs-val s1) (get-grammar-val s1)
                                      (get-store2-store sigma))))
          ...
          (else (error '_o "unknown primop: ~s" op)))))
```

Figure 7.1: The procedures of the implementation corresponding to the valuation functions.

```
; Id* x PTemplate -> Penv
(define (mk-penv names ptemplates)
    (letrec ((tau (lambda (f)
                    (mk-proc f (list-ref ptemplates (find-position f names))
                             tau))))
        (cons tau cproc-init)))

; FId x Penv ->  Proc
(define (lookup-proc f penv)
    (let ((try (lookup-cenv f (cdr penv))))
        (if (equal? try 'undef-cproc)
            ((car penv) f) (cdr penv))
            try)))

; PName x PTemplate x Template -> CEnv -> Val* x Store -> Result
(define (mk-proc name ptemplate tau)
  (lambda (phic)
    (lambda (vals sigma)
      (let ((alpha (mk-non-term-proc name sigma))
            (args (generalize-args vals))
            (sigmap (generalize-store sigma)))
        (let ((res1 (fix-res
                     (lambda (res1)
                       ((ptemplate
                         (cons tau
                               (extend-phic name args
                                            (lambda (vals sigma)
                                              (abstract-result alpha res1))
                                            phic)))
                        args sigmap)))))
          (abstract-result alpha res1)))))))
```

Figure 7.2: The procedures of the implementation corresponding to the combinators.

definitions. Note, though, that in `abstract-result`, we have taken advantage of the fact that the first component of the store (the "seenbefore" list) does not affect the output in the grammar-generating semantics. So we can simplify the output by not abstracting this component of the store.

```
; Def -> Penv -> Val* x Store -> Result
(define (mk-ptemplate def)
    (let ((ides (get-ides-def def))
          (expr (get-expr-def def)))
        (lambda (phi)
            (lambda (vals sigma)
                ((_e expr (extend-env ides vals env-init) phi) sigma)))))

; String* x Result -> Result
(define (abstract-result r v)
    (let ((a1 (car r))
          (a2 (cadr r))
          (a3 (caddr r))
          (v1 (get-val-res v))
          (s1 (get-store1-res v))
          (s2 (get-store2-res v)))
        (mk-res (abstract-value a1 v1)
                (mk-store s1
                          (abstract-value a3 s2)))))

; Non-Term x Val -> Val
(define (abstract-value n v)
    (let ((r (get-rhs-val v))
          (g (get-grammar-val v)))
        (mk-val (mk-rhs (mk-non-term-proc-alternate n))
                (insert-rule (mk-rule n r) g))))

; Val x Val -> Val
(define (val-merge v1 v2)
    (let ((r1 (get-rhs-val v1))
          (r2 (get-rhs-val v2))
          (g1 (get-grammar-val v1))
          (g2 (get-grammar-val v2)))
        (mk-val (rhs-union r1 r2)
                (grammar-union g1 g2))))

; Val* -> Val*
(define (generalize-args v*)
    (map generalize-arg v*))

; Val -> Val
(define (generalize-arg val)
    (replace-by-val val))
```

Figure 7.3: The implementation of some auxiliary definitions.

Chapter 8

Extensions and Improvements

8.1 Complexity Improvements or Better Ways of Doing the Same

As it stands, the analysis does not have a practical complexity for large examples. There are several ways that this could be improved without changing the basic design of the algorithm.

There are three main sources of inefficiency in the algorithm:

- The algorithm computes a fixed point every time it analyzes a procedure.
- The memo-function is only propagated downwards in calls, so the same procedure will be analyzed several times if it is called for example in both branches of a conditional.
- The abstract domain of values is quite large and the individual values in the domain are quite complex, taking up a lot of space and complicating equality tests.

The inefficiencies from these sources multiply, making it even more interesting to remove them.

Fortunately, we observe that the fixed point at the application of procedures does not have much practical purpose. Since we do not access the right hand sides of productions, it does not make any difference in the result if a non-terminal is bound to bottom, as long as the production is there, and so the fixed point is in fact reached in one iteration. The only purpose in introducing the fixed point in the first place is to establish the relation to the concrete interpretation as a local relation on the domain of values. This means that we can simply remove the local fixed points from the algorithm[1].

On a similar note, we observe that when we generate non-terminals for the parameters, the re-evaluation of a procedure with different (ground) arguments is in fact unnecessary. We generate non-terminals for each of the arguments and different arguments extend the productions for these non-terminals. Since we assume that no operations access the right hand sides of a production, the result of a procedure evaluated with a new set of arguments is the same as before, only with the productions for the arguments extended. Thus there is really no need to re-evaluate the procedure at all, it is sufficient to extend the productions for the arguments. Note, however, that in the higher-order case, for lambda abstractions with free variables, the situation is slightly more complicated. In order to avoid re-evaluating the abstraction with a new environment, we would have to generate

[1] We would of course need a formal proof that the two algorithms were equivalent, based on the reasoning outlined above.

non-terminals for the free variables, in the same fashion as we do for the arguments. This, in a sense, corresponds to a kind of lambda lifting "on the fly", which is not surprising, since lambda-lifting would be another way of handling the higher-order values.

The problem of the memo-function is most easily solved by globalizing it. This extra step could be proven safe by writing an intermediary semantics, that contains both a local and a globalized memo-function. In this semantics, we could then prove that the two memo-functions contain identical entries (appealing to extensionality). Then it is of course safe to discard the local memo-function.

We could simplify many of the operations on the domain of values by globalizing the grammar part of the domain. This would mean that the abstraction of a value would now be only (a list of closures and) a right hand side. This is clearly an "underspecified" value, only in connection with the (now globalized) grammar can it be compared to a concrete value. Such a change would clearly simplify an implementation but complicate the safety proof. Again, it would probably be the easiest to prove safety via an intermediary semantics, using both the local and the globalized grammar. A rule in the globalized grammar would be greater than or equal to the rule for the same non-terminal in the local grammar. Thus it would be safe (but potentially cause a loss of information) to discard the local grammar.

A smaller, but similar, modification would be to globalize the environment of closure environments (this would make the analysis closer to the one of Shivers [Shi91]). This could completely remove environments from closures and the environment would no longer contain a closure environment component. Since different environments might cause the closure to evaluate to different results, it would be necessary to generate non-terminals for the free variables of the closure and to update these along with the changes to the (now global) environment of closure environments.

We would not be able to use Shiver's simple 0CFA scheme, though, since too much information would be lost in the typical higher-order generating extension.

8.2 Extensions or Doing More

8.2.1 Improving the precision

A significant improvement of precision could be obtained with a more precise interpretation of the primitive operations that access the right hand sides of productions. However, much of our reasoning (including termination) is based on the assumption that this does not happen! Clearly it is possible to provide some limited access without violating this reasoning, but much delicacy is needed. It would, for example, be possible to allow destructors and predicates on well-founded algebras to access the right hand sides without compromising termination, but clearly not constructors unless the algebra was finite. Instead, we could rely on the abstraction of procedures and procedure parameters to limit constructions and possibly prove termination this way.

Such improvements would also invalidate most of the speedup improvements suggested above.

8.2.2 Removing the restriction on closures

So far we have assumed that closures could not be part of data structures. This is because closures would appear at the right hand side of productions. Since we do not access the right hand sides, this corresponds to losing all information about the closure.

In order to remove this restriction on closures, we could probably use the observation that for any given program, there will only be a finite number of closures. This, combined with the extensions of primitive operations to access right hand sides proposed above, should make it possible to allow

closures in for example lists. Note that if we keep the closures separated on the right hand sides, we can let the primitive destructors access a list of closures but still return Val on a list of ground values.

Another possibility would be to add a separate closure analysis (capable of handling closures in data structures) in a pre-processing phase, annotating all application points. The grammar-generating analysis could then ignore closures and operate on ground values only. Note, however, that this method would not immediately give the same level of precision in the closure environments as we have here.

8.2.3 Distinguishing between call-sites

As mentioned in chapter 4, we could also improve precision by generating more than one non-terminal for each procedure. A promising option would be to generate a separate grammar rule for each call-site. This is easily obtained by adding labels in the program corresponding to the call-sites. A proof of correctness would be almost identical to the one we have already (there is nothing conceptually different in this idea), but the analysis would be more complicated and give more verbose output. It is quite likely that it would be worth the trouble in the higher-order case, though: the typical generating extension produced by higher-order Similix wraps the same lambda abstraction around most of its constructions and often applies the same procedure to different constant arguments. By separating call-sites, these can be kept separate. This effect can also be seen in the first-order case, as exemplified by the rule for `v-PROC-spec-pcall-0-2` in the regular expression example.

8.2.4 Adding information to the grammar

A different kind of extension would be to trace more execution information instead of just computing the abstract values. This information could possibly be represented as attributes to the grammar. This should make it possible to get some information about for example the size of the residual program or (correspondingly) the run time of the generating extension, *i.e.*, the compilation time.

As an example, suppose that we give each of the static data a unique name (say, $s1$, $s2$, *etc.*, if these names are not used). If a procedure in the generating extension takes the car of the first static argument, the analysis currently returns **Val**. Now we could add the attribute that it was actually car of $s1$. Suppose the procedure now calls itself with the cdr of $s1$. Then we can add as an attribute to the corresponding parameter that it is the cdr of $s1$. If we consider the appending program of section 3.3.3, we could get an attributed grammar as in figure 8.1. In this example, it is quite clear how the attributes were determined from the generating extension in figure 3.5. The attributes essentially correspond to those parameters that are bound to static data.

Once the grammar is decorated with attributes, we can use that information to conclude that the size of the residual programs in this case will be linear in the sum of the two static lists. (We would still need a proof of safety for this extension, though.)

8.3 Modifications or Other Ways of Doing the Same

8.3.1 Selecting different program points

In the spirit of Similix, let us note that it would be possible to choose other program points than procedures and lambda abstractions to generate non-terminals. Appropriate candidates include cons points (as used by Consel [Con90]), application points, and conditionals (the latter would

specialize-$0_{s1,s2}$	::=	**(define (ID-main ID-z)** **(cons** expr_{s1} expr_{s2}**))**
expr_s	::=	ID-z_{nil} | **(cons** $\text{Cst}_{(cars)}$ $\text{expr}_{(cdrs)}$**)**

Figure 8.1: Attributed BNF describing the possible results of specializing the program of figure 3.3 with the two first arguments static and a dynamic third argument

correspond to the Similix choice). After all, the purpose of the non-terminals is to break loops in the call graph or, in other words, "fold" the potentially infinite execution graph of the program. This could as well be done by breaking at the conditionals (unless of course there is an unconditional loop).

Note that with a generous selection of program points, some of our termination reasoning could be simplified. If we for example generate a non-terminal at each construction point (application of a primitive constructor), we have a restriction on the size of the alternates, since they can only contain one constructor. This way, we could probably prove that only a finite set of grammars was available for any given program (finite sets of non-terminals, finite size of alternates and a finite number of alternates in each rule). This would allow us more freedom in chosing abstract versions of destructors, since we could fall back on ordinary termination reasoning. This would be similar to the approach of Mogensen [Mog88]. This approach would probably lead to exceedingly verbose output in our case, though, since we have so many constructors and since they are often used in trivial combinations.

8.3.2 Analyzing the binding-time analyzed programs directly

For a different approach to the problem of detecting information about residual programs in advance, it is likely that this analysis could be "pulled back" over the partial evaluator. This would mean analyzing the binding-time annotated source programs directly, instead of the generating extension. The results should be quite similar, but it would be much more complicated to prove that the analysis was consistent with the partial evaluator, since the proof would have to involve the behavior of the partial evaluator instead of just the standard semantics of the target language.

Alternatively, an analysis directly on binding-time analyzed programs could be proven correct by generating it from the present analysis by partial evaluation. Since the present analysis works on the generating extension, it can be described as a function on the partial evaluator and the source program that first produces a generating extension and then analyzes it. If we denote the program analyzing generating extensions by A and the partial evaluator by PE and the functions they compute by $(\!|A|\!)$ and by $(\!|PE|\!)$, as before, we can describe the composed program Γ as a mapping from a partial evaluator and a binding-time analyzed source program to a grammar:

$$(\!|\Gamma|\!)\,[PE, p] = (\!|A|\!)[(\!|PE|\!)[PE, p]]$$

Clearly this composed program can be specialized with respect to its first argument:

$$(\!|PE|\!)[\Gamma, PE]$$

The specialized composed program will be a program that takes a binding-time analyzed program and produces a grammar representation of the possible residual programs. Note, however, that

this is only an extensional achievement. To obtain non-trivial results from this specialization, we would need a more powerful partial evaluator than the current state of the art provides, since the generating extension is produced under dynamic control. Intensionally, the resulting analyzer would in fact operate by first producing the generating extension and then analyzing it.

It would also be possible to specialize the composed program by hand, though, in which case we have a much richer collection of meaning-preserving transformations at our disposal. Thus it may be possible to derive an analyzer operating directly on the binding-time analyzed program in a non-trivial way by transforming the current analyzer. If we only use meaning-preserving transformations and if we have proven that the program *PE* is indeed a partial evaluator, the correctness of such an analysis will follow from the correctness of the current generating extension analyzer.

Chapter 9

Related work

Most of the related work has already been mentioned in chapter 2. In this section we will focus on comparisons with the present approach.

9.1 Partial Evaluation

Some partial evaluators have been proven to produce residual programs with some determinable intensional properties, *e.g.*, in Similix, computations in the source program are guaranteed not to be duplicated in the residual program [BD91].

The λ-Mix partial evaluator has been formally proven to fulfill the definitional condition of a partial evaluator [Gom89].

These properties differ from the present approach by being fixed properties of the partial evaluator, that can be determined once and for all.

Except for the present work, two other analyses concerning the quality of partial evaluation results have been developed recently. One is the analysis tools for aiding the rewriting of source programs to specialize better (the so-called binding-time debuggers). The other is the complexity-related work on speedup and partial evaluation, and the related speedup analysis.

9.1.1 Binding time debuggers

In the partial evaluation community, the actual performance of the residual programs has mainly been considered as an issue for extending the partial evaluator to be more "clever" [Har77, FN88] or for rewriting the source program to "specialize better" [Mal89, Bon91b]. There has been the tacit understanding that knowing the principles of how a partial evaluator works makes it possible to write the source program so that it specializes well. Some work has also been done on finding heuristics for automatically rewriting a program to specialize well [CD91, Bon92].

To aid this process, binding time analysis tools have been added to both Schism [CP92] and Similix [Mos91] recently. These tools can be used to determine why some part of a source program is classified as dynamic by the binding time analyzer, *i.e.*, which dynamic value flows into it. If this does not match the user's idea of which parts of the program ought to be specialized, and if the user understands the basic algorithm used by the binding time analyzer, this information might be used to rewrite the program to specialize better.

So they are a kind of debugging tool or trace mechanism for the binding-time analysis (thus the name).

In some sense these tools are comparable to our present goal, but they aim at the source program, and only give indirect information about the residual program, implied by an assumption about what the partial evaluator will do given a particular annotation.

We aim to produce direct, explicit results about the residual program, that can be used without any particular understanding of the partial evaluation algorithm and that are formally correct without any assumptions about the behaviour of the specializer.

9.1.2 Speed-up analysis

In [AG92] Andersen and Gomard presents a speed-up analysis for partial evaluators for simple imperative programs with loops. This analysis is coupled with a theoretical result that a partial evaluator that uses only constant folding and program point specialization can at most achieve linear speed-up. This result relies on giving a precise and meaningful definition of how to measure the speed-up achieved by partial evaluation.

These results are extended and simplified in [JGS93].

The speed-up analysis differs from the present work in several ways. In both cases, the task is to obtain information about the residual program from the binding-time analyzed source program.

However, the result of speed-up analysis is a numeric interval (possibly using ∞), that the speed-up will approach for sufficiently large dynamic data. There is no information about any other intensional properties.

The result of the present analysis is a grammar describing the family of residual programs. The grammar does not give any speed-up information, but rather structural information. A possible extension of the grammar to contain speed-up information would give complexity estimates for the residual program as a function of the size of the static data, rather than the relative speed-up compared to the source program, given as a fixed number.

9.2 Abstract Interpretation

The present work depends strongly on previous work in abstract interpretation.

Our abstract interpretation is based on denotational semantics, as suggested by Nielson [Nie82, Nie89]. In the proof of correctness, we use the technique of logical relations as described by Mycroft and Jones [JM86b] and Nielson [Nie89].

The representation of abstract information as grammars and the schema of having non-terminals for procedures and parameters was inspired by previous work on grammar-generating interpretations, in particular Jones [JM86a, Jon87] and Mogensen [Mog88].

Our analysis differs from this work in two respects: by the absence a of global fixed point and by treating procedure calls depth-first. These are in fact related, the global fixed point is unnecessary because calls are treated depth-first, so when the analysis returns from the initial call, there will be no un-processed calls to analyze.

The depth-first treatment of procedure calls is comparable to the development in partial evaluation, where the breadth-first strategy of the original Mix [JSS89] was replaced by a depth-first strategy in later partial evaluators, such as Similix and Schism.

In the first-order case, where we are able to eliminate the local fixed-points, the depth-first strategy has the advantage of a more "predictable" complexity, and the elimination of the "extra" iteration that is needed to determine that the fixed point has been reached. The elimination of the fixed point relies on properties that are particularly tailored to this problem, though, so we cannot immediately expect this to carry over to other applications. We are, for example, not seriously

hampered by trivializing all destructors, since we are mainly interested in residual expressions, and there are no destructors on residual expressions. This will generally not hold: even as soon as we move to higher order, it imposes a restriction on the use of closures.

9.2.1 Closure analysis

Our handling of higher-order procedures is based on closure analysis and is strongly influenced by the work of Shivers [Shi91] and Sestoft [Ses91].

Shivers classifies different kinds of closure analyses into 0CFA and 1CFA according to the handling of closure environments.

A 0CFA analysis uses a global cache to hold the least upper bound of the environments encountered for each of the lambdas.

A 1CFA analysis also use a global cache, but distinguishes not only between different lambdas, but also between different call-sites. So the type of the cache is $\mathit{Program\text{-}Point} \times \mathit{Call\text{-}Site} \rightarrow \mathit{Env}$.

The environment mechanism that we use in the higher-order analysis differs from both these by using a local, rather than a global, strategy to ensure finiteness, with EnvEnv_γ corresponding to the global cache. The local handling of environments allows for more detail than a 0CFA analysis, since the relationship between values in the closure environment is preserved[1].

On the other hand, our analysis is not quite 1CFA, either, since it does not distinguish between call-sites in the generation of non-terminals. To fully explore the level of detail in the environments, we could generate separate grammar rules for each distinct pair of a program point and an environment, instead of merging the results for all closures with the same program point.

A pure closure analysis removes all concrete values from the abstract domains and traces the flow of the higher-order values. Our analysis integrates the closure analysis with the grammar-generating analysis. This would make the loss of information in the 0CFA analysis even more noticeable.

This merging of two kinds of analysis is similar to Consel [Con90] who uses an integrated binding-time analysis/closure analysis that also handles data-structures, that is, Lisp lists with the operations car, cdr, cons, and null?. There the abstract domain of values (representing binding-times) contains partially static structures and abstract closures as well as the "ground values" static and dynamic.

The binding-time analysis of Consel does not have our restriction on construction over closures. This does not cause a problem because destructors are allowed on the abstract values. The domain of abstract ground values for binding-time analysis is finite, and even quite small, so the analysis will still terminate in a reasonable amount of time.

Consel's analysis is 0CFA, but the problem of merging contexts that this gives is avoided by rewriting the source program at the points where the analysis discovers such a merging (the so-called polyvariant binding-time analysis).

Bondorf uses a pure closure analysis as a prephase to binding-time analysis, so his analysis is less related to ours. His analysis is a straightforward closure analysis, as Sestoft [Ses91]. It updates a global cache with information for every expression and variable in the program and computes a fixed point over this cache. It only remembers the label of the lambda abstraction for each closure, not the closure environment, so according to the classification of Shivers, it is 0CFA. The binding time analysis then uses this information: at each application point it propagates the binding times of the arguments to all the possible procedures.

[1] If, for example, we have a call `...(f (lambda () (- a b)))...` in a context where `a` and `b` are always bound to the same value, and `f` is defined as `(define (f 1) (1))`, then we can determine that `f` always returns 0, whereas a 0CFA analysis would use all possible combinations of the values of `a` and `b`.

Chapter 10

Conclusion

We have developed a method for generating abstract representations of residual programs from a partial evaluator and a source program. We obtain a grammar describing the family of residual programs by first producing a generating extension and then performing an abstract interpretation. The abstract interpretation ensures termination by replacing recursive functions by constant functions returning recursive values.

We have shown that the abstract interpretation is safe in the sense that any concrete result is guaranteed to be in the language represented by the abstract result. The safety is proven using logical relations.

Several improvements (in terms of complexity) and extensions (in terms of precision) have been suggested – generally counteracting each other.

We have implemented a prototype of the analysis in Scheme and run several examples to demonstrate the kind of information we get from the analysis.

Our abstract interpretation is also useful in more general cases, as shown by the example of the CPS transformer.

In applications of partial evaluation, it is usually claimed in an informal way that residual programs obtained by specializing a given source program have a particular structure. Our analysis gives a method for formally justifying such claims.

Part of the reason for making such claims is the theoretical interest in showing that residual programs will have the natural or efficient structure provided by some known algorithm. But it is potentially of larger significance to know that the residual programs will always be restricted to a few simple forms. This enables fast implementation of families of residual programs by (automatically?) taking advantage of the restrictions. Our analysis automatically provides tight restrictions on the format of residual programs, that could clearly be used in such a strategy.

Bibliography

[AG92] L. O. Andersen and C. K. Gomard. Speedup analysis in partial evaluation (preliminary results). In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University)*, pages 1–7, 1992.

[App92] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[Bar91] G. Barzdins. Iterative properties of partial evaluation. Technical report, Computer Science Department, New Mexico State University, 1991.

[BD91] A. Bondorf and O. Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[BE88] M. A. Bulyonkov and A. P. Ershov. How do ad-hoc compiler constructs appear in universal mixed computation processes? In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 65–81. North-Holland, 1988.

[Bon90] A. Bondorf. *Self-Applicable Partial Evaluation.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990. Revised version: DIKU Report 90/17.

[Bon91a] A. Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, 1991.

[Bon91b] A. Bondorf. Similix manual, system version 4.0. DIKU, University of Copenhagen, Denmark. September, 1991.

[Bon92] A. Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.

[Bul84] M. A. Bulyonkov. Polyvariant mixed computation for analyzer programs. *Acta Informatica*, 21:473–484, 1984.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[CD89] C. Consel and O. Danvy. Partial evaluation of pattern matching in strings. *Information Processing Letters*, 30:79–86, January 1989.

[CD90a] C. Consel and O. Danvy. From interpreting to compiling binding times. In N. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 88–105. Springer-Verlag, 1990.

[CD90b] C. Consel and O. Danvy. Partial evaluation in parallel (detailed abstract). Research Report 820, Computer Science Department, Yale University, 1990.

[CD91] C. Consel and O. Danvy. For a better support of static data flow. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 496–519. ACM, Springer-Verlag, 1991.

[CK91] C. Consel and S.C. Khoo. Parameterized partial evaluation. In *Sigplan '91 Conference on Programming Language Design and Implementation, June 1991, Toronto, Canada*, pages 92–106. ACM, 1991.

[Con90] C. Consel. Binding time analysis for higher order untyped functional languages. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 264–272. ACM, 1990.

[CP92] C. Consel and S. Pai. A programming environment for binding-time based partial evaluators. In *Partial Evaluation and Semantics-Based Program Manipulation, San Francisco, California, June 1992. (Technical Report YALEU/DCS/RR-909, Yale University)*, pages 62–66, 1992.

[CR91] W. Clinger and J. Rees. Revised[4] report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[Dan91] O. Danvy. Semantics-directed compilation of non-linear patterns. *Information Processing Letters*, 37:315–322, March 1991.

[Dan92] O. Danvy. Back to direct style. In B. Krieg-Brückner, editor, *Proceedings of the Fourth European Symposium on Programming*, number 582 in Lecture Notes in Computer Science, pages 130–150, Rennes, France, February 1992.

[Dan93] O. Danvy. Back to direct style. *Science of Computer Programming*, 1993. Special issue on ESOP'92, the Fourth European Symposium on Programming, Rennes, February 26-28, 1992. To appear.

[DF92] O. Danvy and A. Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, 1992.

[Dyb85] H. Dybkjær. Parsers and partial evaluation: An experiment. Student Project 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985. 128 pages.

[Ell80] P. Emanuelson and A. Haraldsson. On compiling embedded languages in Lisp. In *1980 Lisp Conference, Stanford, California*, pages 208–215, 1980.

[Ema80] P. Emanuelson. Performance enhancement in a well-structured pattern matcher through partial evaluation. Linköping Studies in Science and Technology Dissertations 55, Linköping University, Sweden, 1980.

[Ema82] P. Emanuelson. From abstract model to efficient compilation of patterns. In M. Dezani-Ciancaglini and U. Montanari, editors, *International Symposium on Programming, 5th Colloquium, Turin, Italy. (Lecture Notes in Computer Science, vol. 137)*, pages 91–104. Springer-Verlag, 1982.

[Ers78] A. P. Ershov. On the essence of compilation. In E.J. Neuhold, editor, *Formal Description of Programming Concepts*, pages 391–420. North-Holland, 1978.

[FL91] P. Fradet and D. Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13:21–51, 1991.

[FN88] Y. Futamura and K. Nogi. Generalized partial computation. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 133–151. North-Holland, 1988.

[Fut71] Y. Futamura. Partial evaluation of computation process – an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[GJ89] C. K. Gomard and N. D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[Gom89] C. K. Gomard. Higher order partial evaluation – HOPE for the lambda calculus. Master's thesis, DIKU, University of Copenhagen, Denmark, September 1989.

[Gom90] C. K. Gomard. Partial type inference for untyped functional programs. In *1990 ACM Conference on Lisp and Functional Programming, Nice, France*, pages 282–287. ACM, 1990.

[GS91] C. K. Gomard and P. Sestoft. Globalization and live variables. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 166–177. ACM Press, 1991. Sigplan Notices, vol. 26, number 9.

[Har77] A. Haraldsson. *A Program Manipulation System Based on Partial Evaluation*. PhD thesis, Linköping University, Sweden, 1977. Linköping Studies in Science and Technology Dissertations 14.

[HD89] N. C. K. Holst and O. Danvy. Partial evaluation without a partial evaluator. draft, 1989.

[Hen91] F. Henglein. Efficient type inference for higher-order binding-time analysis. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. ACM, Springer-Verlag, 1991.

[Hol88] N. C. K. Holst. Language triplets: The AMIX approach. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 167–185. North-Holland, 1988.

[JGS93] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993. To appear.

[JM86a] N. D. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Thirteenth ACM Symposium on Principles of Programming Languages, St. Petersburg, Florida*, pages 296–306. ACM, 1986.

[JM86b] N. D. Jones and A. Mycroft. A relational framework for abstract interpretation. In H. Ganzinger and N.D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark. (Lecture Notes in Computer Science, vol. 217)*, pages 156–171. Springer-Verlag, 1986.

[JM90] T. P. Jensen and T. Mogensen. A backwards analysis for compile-time garbage collection. In *ESOP '90, Copenhagen, Denmark (Lecture Notes in Computer Science, vol. 432)*, pages 227–239. Springer-Verlag LNCS 432, 1990.

[Jon87] N. D. Jones. Flow analysis of lazy higher-order functional programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 103–122. Ellis Horwood, Chichester, England, 1987.

[Jon88] N. D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[Jon90] N. D. Jones. Partial evaluation, self-application and types. In M.S. Paterson, editor, *Automata, Languages and Programming. 17th International Colloquium, Warwick, England. (Lecture Notes in Computer Science, vol. 443)*, pages 639–659. Springer-Verlag, 1990.

[JS86] U. Jørring and W. L. Scherlis. Compilers and staging transformations. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 86–96, St. Petersburg, Florida, January 1986.

[JSS85] N. D. Jones, P. Sestoft, and H. Søndergaard. An experiment in partial evaluation: The generation of a compiler generator. In J.-P. Jouannaud, editor, *Rewriting Techniques and Applications, Dijon, France. (Lecture Notes in Computer Science, vol. 202)*, pages 124–140. Springer-Verlag, 1985.

[JSS89] N. D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Jør91] J. Jørgensen. Generating a pattern matching compiler by partial evaluation. In S.L. Peyton Jones, G. Hutton, and C. Kehler Holst, editors, *Functional Programming, Glasgow 1990*, pages 177–195. Springer-Verlag, 1991.

[Kle52] S. C. Kleene. *Introduction to Metamathematics*. D. van Nostrand, 1952.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, June 1977.

[Mal89] K. Malmkjær. Program and data specialization, principles, applications, and self-application. Master's thesis, DIKU, August 1989. 86 p.

[Mog88] T. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Mog89] T. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989. 95 pages.

[Mos91] C. Mossin. Similix binding time debugger manual, system version 4.0. Included in Similix distribution, September 1991.

[Myc81] A. Mycroft. *Abstract interpretation and optimizing transformations for applicative programs*. PhD thesis, University of Edinburgh, Scotland, 1981.

[Nie82] F. Nielson. A denotational framework for data flow analysis. *Acta Informatica*, 18:265-287, 1982.

[Nie89] F. Nielson. Two-level semantics and abstract interpretation. *Theoretical Computer Science*, 69(2):117-242, 1989.

[NN86] F. Nielson and H. Riis Nielson. Semantics directed compiling for functional languages. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, 1986.

[NN88] F. Nielson and H. Riis Nielson. Two-level semantics and code generation. *Theoretical Computer Science*, 56(1):59-133, January 1988.

[NN92] F. Nielson and H. Riis Nielson. The tensor product in wadler's analysis of lists. In *ESOP '92. 4th European Symposium on Programming, Rennes, France, February 1992. (Lecture Notes in Computer Science, vol. 582)*, 1992.

[Plo75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1:125-159, 1975.

[Plo83] G. D. Plotkin. Domains. unpublished course notes, 1983.

[Rom88] S. A. Romanenko. A compiler generator produced by a self-applicable specializer can have a surprisingly natural and understandable structure. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 445-463. North-Holland, 1988.

[Sch86] D. A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1986.

[Ses86] P. Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236-256. Springer-Verlag, 1986.

[Ses91] P. Sestoft. *Analysis and efficient implementation of functional programs*. PhD thesis, Computer Science Department, University of Copenhagen, October 1991.

[Shi88] O. Shivers. Control flow analysis in Scheme. *Sigplan Notices*, 23(7):164-174, July 1988. Sigplan Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 1988.

[Shi91] O. Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie Mellon, May 1991. CMU Technical Report CMU-CS-91-145.

[Ste78] G. L. Steele, Jr. Rabbit: A compiler for Scheme. Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.

[Tal85] C. L. Talcott. *The Essence of Rum: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*. PhD thesis, Department of Computer Science, Stanford University, Stanford, California, August 1985.

[Tur79] V. F. Turchin. A supercompiler system based on the language Refal. *Sigplan Notices*, 14(2):46-54, February 1979.

[Tur86a] V. F. Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292-325, July 1986.

[Tur86b] V. F. Turchin. Program transformation by supercompilation. In H. Ganzinger and N. D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 257-281. Springer-Verlag, 1986.

[Tur88] V. F. Turchin. The algorithm of generalization in the supercompiler. In D. Bjørner, A.P. Ershov, and N.D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 531-549. North-Holland, 1988.

[Tur89] V. F. Turchin. *Refal-5, Programming Guide & Reference Manual.* New England Publishing Co., Holyoke, Massachusetts, 1989.

[Wan82] M. Wand. Deriving target code as a representation of continuation semantics. *ACM Transactions on Programming Languages and Systems*, 4(3):496-517, 1982.

[Wei73] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the IEEE Symposium on Switching and Automata Theory*, 1973.

Abstract

Partial evaluation is an automatic program transformation technique that instantiates general programs to specialized programs, enabling removal of interpretive overhead and even automatic compiler generation. Partial evaluators are defined by an extensional correctness condition: running the partial evaluator on the source program and part of its input data and then running the specialized program on the remaining data must give the same result as running the source program on all the data (modulo termination). So the definition of partial evaluation guarantees that the specialized programs will be correct, that is, that they will compute the right function. Computer scientists, however, are not only interested in which function a program computes, but also in how it computes it: the intensional aspects. The definition of partial evaluation is purely extensional and does not give any guarantees about the intensional properties of the specialized programs. In order to fully exploit the potentials of partial evaluation, this aspect must also be explored, preferably with the same solid foundation in formal correctness as the extensional definition gives.

In the general framework of partial evaluation of applicative programs based on polyvariant specialization, we study the family of specialized programs that can be generated from a given general program. We consider which intensional properties the programs in such a family will share and whether they can be pre-determined. We develop an analysis, based on abstract interpretation, that generates a grammar describing the family, based only on the partial evaluator and (parts of) the source program. We prove that the analysis is safe with respect to a standard semantics, in the sense that a concrete result is in the language generated by the abstract result. We use the analysis on several examples and show that various informal claims about the intensional properties of specialized programs can be justified formally. The analysis is also generally applicable to other program transformers and we show an example of this.