Predicting Properties of Specialized Programs

PhD Proposal

Karoline Malmkjær

November 20, 1991

Contents

Chapter 1

Introduction

Partial evaluation is a program transformation principle that is defined in terms of an extensional correctness criterion. In practical applications it is used in the hope of obtaining some level of optimization, but this optimization is not guaranteed by the definition.

The present work investigates how to determine intensional properties of programs that are results of such a program transformation before the transformation actually takes place.

In partial evaluation a program is specialized with respect to part of its input. A partial evaluator takes a source program and part of the input data of the source program and produces a specialized program with the property that the specialized program run on the rest of the data gives the same result as the source program run on the complete data. Thus partial evaluation is a technique for deriving formally correct programs easily. If the partial evaluator is correct, which can be proven once and for all, and if the source program is correct or definitional, the specialized program is formally correct. This correctness follows from the definition of a partial evaluator and is purely extensional.

While correctness is a strong argument for using partial evaluation as a general program development tool, the potential applications of partial evaluation in areas such as compiler generation require not only that the residual program is extensionally correct – it must also fulfill a number of intensional conditions. Examples of such conditions on residual programs include restrictions on the use of language constructs, structural properties such as single-threadedness in certain variables or tail-recursion, efficient use of storage both in terms of quantity and allocation strategy, and even such simple things as bounds on size.

So far these issues have only been considered informally in the partial evaluation community. There has been more than enough problems in developing practical partial evaluators for a reasonable set of reasonably complex languages. The actual performance of the residual programs has been considered only as an issue for extending the partial evaluator to be more "clever" [Haraldsson 77, Futamura & Nogi 88] or for rewriting the source program to "specialize well". There has been the tacit understanding that knowing the principles of how a partial evaluator works allow you to write your source program so that it specializes well. Some work has also been done on finding heuristics for automatically rewriting a program to specialize well [Consel & Danvy 91, Holst & Gomard 91].

In order to improve understanding of the partial evaluation process, there have been several studies on the algorithms produced when specializing a specific program with respect to an arbitrary argument. Several of the MIX papers discuss the structure of target programs obtained by partial evaluation [Sestoft 86, Gomard & Jones 89, Jones *et al.* 89] and Consel and Danvy investigate the structure of residual programs obtained by specializing different string matching algorithms in [Consel & Danvy 89].

But so far there have been no attempts to develop tools that would determine automatically and correctly what the results of specializing a specific program would look like.

This goal is not as unapproachable as it may sound at first. Although existing partial evaluators become increasingly complex, the basic algorithm of off-line polyvariant specialization [Bulyonkov 84] is characterized by its regularity. Based on a small number of parameters it selects one of a small number of actions, over and over. As output it produces a collection of source program pieces that have been determined since the binding time analysis and are glued together in a structure determined from the processing of the static data.

We use this knowledge of binding time based regularity when we rewrite source programs to specialize well: write the code just so and this piece will always be eliminated and that piece will always be reconstructed. So it

is not unreasonable to expect that given for example 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. Properties such as whether all calls are tail-recursive, which variables are strict, size or storage use expressed as functions of the size and structure of the static data.

1.1 Outline

The next chapter gives a brief introduction to partial evaluation and to abstract interpretation. Anyone familiar with these areas can go directly to chapter 3, that presents the problem we are dealing with in some more detail and presents various approaches to a solution. Chapter 4 goes into more detail with the most promising of these approaches, based on a grammar-generating non-standard interpretation of the generating extension. Chapter 5 summarizes what has been done and outlines further work on the project.

1.2 Terminology and notation

This section introduces some terminology and notational conventions used in the following chapters. Some of these originate in the partial evaluation community while others are introduced here for clarity.

We will distinguish between the *extensional* and the *intensional* properties of programs, as defined by Carolyn Talcott [Talcott 85]. This is a computer science equivalent of the mathematical usage, where the extensional properties concern which function a program computes, while the intensional properties concern its text and execution.

In partial evaluation, the notion of *binding time* has great significance. The usual compiler concept of binding time for variables is extended, so that also expressions and computations are assigned a binding time. We usually distinguish between at least two binding times, called static and dynamic, that correspond roughly to the usual notions of compile time and run time. The distinction also relates to the distinction between static and dynamic typing.

When reasoning about the manipulation of programs by programs, some confusion between references to the program text and references to the action performed, when the program is run, appears to be inevitable. In this paper we will adopt the following notational conventions in order to minimize the confusion: actual program text is always shown in `typewriter` font, meta-level, semantic, or mathematical names are usually shown in *italics*. If a meta-variable *program* denotes the program `(define (foo l) (if (null? l) 0 (add1 (foo (cdr l)))))`, then a reference to *program* is a reference to the text: `(define (foo l) (if (null? l) 0 (add1 (foo (cdr l)))))`. A reference to the function this program computes when run in some programming language L is denoted $[\![program]\!]_L$, so $[\![program]\!]_{Scheme} = length$. A refence to `foo`, on the other hand, is simply a reference to the 3-letter identifier `foo`.

We will use the term *function* only for mathematical functions (including terms of the λ-calculus), while a piece of program that can be invoked somewhere else in the program by referring to its name will be referred to as a *procedure*. Thus we will say that the procedure named `foo`, defined in *program*, computes the function *length*.

Function application will be denoted simply by juxtaposition of the function and the argument, possibly surrounded by parentheses. For lists and tuples we will use square brackets.

We will use the word *domain* in the meaning *complete partial order*. When we have a domain D, we use D^* to denote the domain of finite tuples over D, ordered pointwise. If we have a tuple $[d_1, \ldots, d_n]$ in D^*, $d :: [d_1, \ldots, d_n]$ denotes the tuple $[d, d_1, \ldots, d_n]$ and $[d_1, \ldots, d_n] \downarrow i$ denotes d_i, the i'th element, if $1 \leq i \leq n$ and $\perp_{D^*}$ if $i > n$ or $i < 1$.

Examples of programs will mainly be written in Scheme [Rees & Clinger 86].

Except when otherwise stated, all specialized programs presented are obtained with the Similix-2 partial evaluator [Bondorf & Danvy 91, Bondorf 91a].

Chapter 2

Background

2.1 Program specialization

In this section we give a brief outline of the field of program specialization, its theoretical founding 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

Program specialization is the process of taking a program and the values of some of its input and producing 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 also called a *partial evaluator*. If we have a partial evaluator *MIX* written in L, the definitional property can be expressed in the following way:

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

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 [1], where the bitmap representing the picture of the moon is "compiled in"[2] to the program loading the bitmap.

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 *[Kleene 52]*
For all m, n there exists a 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.

[1] X Window System is a trademark of the Massachusetts Institute of Technology.
[2] to quote the xphoon manual page

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[3]

```
(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))))
```

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 be 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. Avoiding this is undecidable, but practical approximations (safe or – more commonly – unsafe) can be made [Jones 88].

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

The λ-mix algorithm

[Gomard 89] 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 [Nielson & Nielson 86].

Polyvariant specialization

One of the most popular algorithms for program specialization is known as *polyvariant specialization* [Bulyonkov 84]. 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 that is 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.

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 can be proven quite easily that avoiding this is equivalent to solving the halting problem.

Driving and the Refal system

One of the pioneer systems in partial evaluation is Refal [Turchin 86a, Turchin 86b], 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" [Turchin 88] 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. [4]

[4] 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.

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 [Futamura 71] 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.

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:

$$[\![t]\!]_T\,d = [\![[\![MIX]\!]_L\quad [int,p]]\!]_T\,d = [\![int]\!]_L\,[p, d] = [\![p]\!]_S\,d \tag{2.1}$$

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.

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

We can also note 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 is in [Ershov 78] called a *generating extension* of *int*. 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.

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.

Most of the development in this section was inspired by [Mogensen 89a]. Reduce/rebuild is introduced as a fundamental concept in [Consel & Danvy 90].

Relating binding time analysis to other analyses

Binding time analysis has clear relations to several other analyses. It is usually done as a kind of abstract interpretation similar to for example strictness analysis. But it is also possible to relate it to type inference, Gomard presents a type-inference algorithm for computing binding times [Gomard 90], extended in [Henglein 91].

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 [Nielson & Nielson 86]. 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.

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 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

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.

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 cps transformation [Consel & Danvy 91].

2.1.7 A polyvariant specializer: Similix

Similix [Bondorf & Danvy 91, Bondorf 91a] is a Mix-style polyvariant specializer for a subset of Scheme, 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[5].

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 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.

[5]This BNF was borrowed from the Similix manual [Bondorf 91a]

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.

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.

2.2.2 Formalization: Cousot and Cousot

The data flow analyses used in compilers were formalized by Cousot and Cousot [Cousot & Cousot 77]. The formalization covered imperative languages and were based on a semantics for the language being analyzed. The semantics was first generalized to sets of states[6] and the generalization was then abstracted to finite abstractions of these sets, so that the analysis would terminate. The correctness of the analysis was 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 $\gamma(f^{\#}(\alpha(s)) \supseteq \{f(v) \mid v \in s\}$.

2.2.3 Relational approach

This work was extended to the general framework of denotational semantics by Nielson [Nielson 82], making it possible to treat a much larger class of languages.

Also the safety proof was simplified by *factorizing* the semantics into a *core* using a set of combinators and an *interpretation*, defining the combinators. By using the same core and only abstracting the combinators, the abstraction can be expressed more simply and its safety can be proven by local reasoning for each combinator instead of a proof involving the entire semantics.

The desired safety condition is expressed as a relation between the abstract and the concrete domains. The relation is extended in the usual way to the compound domains. For each combinator of type T, it must then be proven that the T relation holds between the standard and the abstract interpretations of the combinator. This is sufficient to show that the relation holds between the standard and the abstract semantics.

2.2.4 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

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

interpretation. In order to stretch these restraints as much as possible, [Jones 87] 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 supposed to represent the result of procedure applications and identifier references. This way any recursive references will appear as recursive productions in the grammar.

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 [Mogensen 88]. 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.

A grammar representation of contexts is used with a backwards analysis to determine liveness by Jensen [Jensen 90].

A backwards analysis with a grammar representation of definition-use paths is used by Gomard and Sestoft to detect variables that can be globalized [Gomard & Sestoft 91].

2.3 Related work on predicting results of program transformation

2.3.1 Partial evaluation

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

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

The λ-Mix partial evaluator has been formally proven to fulfill the definitional condition of a partial evaluator [Gomard 89].

Recently, binding time analysis tools have been added to both Schism [Consel 88, Consel 90] and Similix. If the user understands the basic algorithm used in these partial evaluators (off-line polyvariant specialization), these tools can be used to determine which parts of a source program will remain after specialization. In that way these tools are comparable to our present goal, but we aim to produce much more explicit results, that can be used without such deep understanding of the partial evaluation algorithm.

2.3.2 Other areas

Outside the area of partial evaluation, there have been some approaches to determining the results of transformations in advance.

As an example, [Ball 79] presents a simple data flow based algorithm for determining whether procedure inlining would be worthwhile, based on execution frequency statistics.

Chapter 3

Predicting Properties of Residual Programs

If both the source program and the static data are given, we can of course determine intensional properties of the residual program by first generating it and then analyzing it, using existing techniques.

On the other hand, if none of the input to a partial evaluator is known, the generic properties of all the residual programs can be determined once and for all. Which target language constructs can the partial evaluator generate, *etc.*

But what about the intermediate cases, for example where 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, corresponding to the infinite set of possible static data. Is it possible to give a finite, intensional description of this family?

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 target programs and we would like to determine properties about its members such as whether all procedure calls are tail-recursive or (in the case of a partial evaluator with the same source and target language) whether they are written in the same subset of the language as the interpreter. Or in the case of an incremental partial evaluation, where the residual program will be specialized again, we would like to make a generic binding time analysis on the whole family, which could determine whether any variable derived from a source variable "x" will always be static or whether any procedure that is a specialization of a source procedure "foo" will always return a dynamic result.

3.1 An example: string matching

In [Consel & Danvy 89], partial evaluation is used to generate programs with the structure of the Knuth-Morris-Pratt algorithm from a naive, two-argument, string matching program. That the specialized programs have the structure of the KMP is shown by example. This idea has been 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 [Weiner 73] in the case where the text is known [Danvy & Malmkjær 91].

Common to all these applications of partial evaluation is that the similarity to known data structures is obtained mainly by an automatic transformation, but that the similarity is only exemplified, not proven for all possible static data.

We would like to have an analysis, that based on the partial evaluator and the string matching source program could tell us whether the residual programs do indeed have the same structure as the known string matching structures.

3.2 Partial evaluation of the analysis

Our problem is that we want to derive properties of the possible residual programs when we know the source program and the binding time pattern but not the static data. Any time we generate one of the residual programs, then we can use an analyzer, based on existing techniques such as abstract interpretation, which can tell us what we want to know for this particular residual program.

Let us try to rephrase this: we have a program (which happens to be composed of two phases, partial evaluation and analysis) and this program expects two arguments, an annotated source program and the static data. But we only have the first of these argument. Clearly the solution is (at least theoretically) obvious: we partially evaluate our composed program with respect to the argument that we do have.

Stated like this, this approach sounds simple, but it has at least two serious problems.

The first is that it is not given that the result will be very useful, since the composed program may not specialize well. In fact a little experience with polyvariant specializers shows that it specializes very badly, since the argument of the second phase is constructed by the first phase under dynamic control.

The second is that the result will be a program and this is not immediately very useful. Some further analysis of this program will be needed to skim the concrete facts about the family.

The approach does not seem very promising at the present, because of the poor performance of the polyvariant algorithm on programs that build intermediate data-structures under dynamic control. The problem is related to Wadler's deforestation techniques [Wadler 88] and essentially concerned with bringing together producers and consumers. Until a solution to this problem is found, the extensional simplicity of this idea does not seem to result in practical performance.

3.3 Analyzing the generating extension

Since it seems complicated to obtain good results by specializing a composition of an analyzer and a partial evaluator, let us try to go "the other way round". Since most polyvariant specializers are designed to be self-applied, we know that the *specializer* at least specializes well, even if its composition with an analyzer does not. So when we have a source program, we can specialize the specializer with respect to this source program, giving the generating extension of the source program. Recall that given the static data the generating extension produces the residual program corresponding to that static data. Extensionally, it is a "syntactically curried" version of the source program: it knows how to specialize the source program and nothing else. Intensionally, it is a specialized version of the specializer and – if the specializer is designed to be self-applicable – it is reasonably well specialized.

The generating extension can itself be considered to be a description of the family of residual programs that we want to characterize, since this family is exactly the set of possible output of the generating extension. As a description, however, it is not very informative, so our task is to design an analyzer which can derive a more suitable description from the generating extension. The correctness of such an analysis can be ensured by formalizing it as an abstract interpretation.

Describing this approach as "going the other way round" is not merely a figure of speech. It is quite likely that the two approaches form a commuting diagram in some suitable setting. Investigating this connection is, however, only a secondary goal for the time being.

The idea of analyzing the generating extension is investigated further in chapter 4.

3.4 Abstract interpretation of the partial evaluation semantics

Instead of first producing the generating extension and then producing an abstract result by abstract interpretation, it is possible that similar results may be obtained by "abstract partial evaluation" of the source program directly. The abstract partial evaluation would be some kind of abstraction of the so-called partial evaluation semantics, in the same way an abstract interpretation abstracts the standard semantics. One difference from the idea of analyzing the generating extension would be that such an approach could be proven safe with respect to the partial evaluation semantics, whereas the generating extension approach can be proven safe with respect to the actual partial evaluator producing the generating extension. In order to prove this approach safe with respect to an actual partial evaluator, one would have to design a separate proof that the partial evaluator implements the partial evaluation semantics, which might make this approach more complicated.

An analysis based on the partial evaluation semantics would probably fit into the parametrized partial evaluation framework of Consel and Khoo [Consel & Khoo 91].

Chapter 4

Analyzing the Generating Extension

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. 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.

To see that this is 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 be sufficient to establish at least some of the intensional properties of the family of residual programs.

4.1 The generating extension as a syntax constructor

Example

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 4.1 takes three lists and appends the third to the end of the first and to the end of the second and then conses the two results together.

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

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

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

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

Figure 4.1: 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)

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 4.1. 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 4.1. Not surprisingly, this program looks pretty much like the first one. Again the calls to append have been unfolded and the static lists decomposed.

```
(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 4.2: Binding time annotated version of the source program in figure 4.1

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

Figure 4.3: BNF describing the possible results of specializing the program of figure 4.1 with the two first arguments static and a dynamic third argument

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 consing some constant values onto the argument and the second of which is the result of consing 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 4.1 gives the annotated program in figure 4.2. 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".

In more precise terms, we would expect it to be in the language generated by the BNF in figure 4.3.

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.

This description actually fits what happens in the generating extensions. Unfortunately generating extensions are usually too large and generally incomprehensible to show[1], but in figure 4.4 we give a simplified pseudo-code version of the generating extension produced by Similix for the source program in figure 4.1.

The BNF of figure 4.3 can be obtained by hand from the generating extension in figure 4.4 by tracing the constructions the generating extension performs.

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

Obtaining this is actually relatively simple in Similix-2, since the same primitive is used to trigger the construction of procedure calls and the construction of procedure definitions. The reasons for this are inherent

[1] 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 4.4: "Pseudo-Similix" generating extension for the program in figure 4.1

in the partial evaluation algorithm used in Similix-2, but need not concern us now. 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 prove that this interpretation of the grammars also preserves the safety relation requires a modification in the language that a grammar is supposed to generate and we have not yet completed this proof.

4.2 Analysis of a first-order language

We have applied the idea of analyzing the generating extension to Similix-2, that is, we have started to design a non-standard interpretation for the language of the generating extensions. As a first approximation, we handle only a first-order subset of this, corresponding to the syntax given in figure 4.5. The non-standard interpretation gives an abstraction of the output of programs in that language.

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

4.2.1 Choosing a suitable abstraction

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

- we are interested in the *output* of the program being analyzed, that is, not in the internal operations of the program, except where this will somehow influence the output. This differs from compiler-oriented applications of abstract interpretation, such as dependency analysis or liveness analysis.
- we are mainly interested in the *structure* of the output, considered as a syntax tree. So it is not very important whether a number is odd or even, or whether it is the result of multiplying two input values. What is important, however, is for example whether a structured value is obtained by applying the primitive `build-cond`.

To represent the structure of an output that may be constructed using recursive calls, we use the technique of abstracting values as grammars [Jones 87]. With this technique, the output of a procedure is denoted by

$P \in Pgm$ $\quad I \in Id$
$E \in Expr$ $\quad O \in Op$
$C \in Cst$ $\quad F \in Proc\text{-}Name$

$$
\begin{array}{lcl}
P & ::= & ((\text{define } (F_1\ I_1 \ldots I_{n_1})\ E_1) \ldots (\text{define } (F_m\ I_1 \ldots I_{n_m})\ E_m)) \\
E & ::= & C \mid I \mid (\text{if } E_0\ E_1\ E_2) \mid (O\ E_1 \ldots E_n) \mid (F\ E_1 \ldots E_n) \mid (\text{let } ((I\ E_1))\ E_2) \\
O & ::= & \text{cons} \mid \text{car} \mid \text{cdr} \mid \ldots \mid \text{\_sim-build-def} \mid \text{\_sim-build-cond} \mid \text{\_sim-generate-proc-name!} \ldots
\end{array}
$$

Figure 4.5: BNF for the first-order subset

a non-terminal and a grammar-rule for that non-terminal describing the construction of the output, possibly using the non-terminal recursively. This provides a finite representation of a possibly infinite output.

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

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

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

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

The following sections give the formalization of these ideas.

4.2.2 Standard and non-standard interpretation

In this section we define the subset of the Similix specializer language that we handle so far. We give the standard semantics of this subset in the form of a core and a standard interpretation. We then give the non-standard interpretation that will generate grammars.

Syntax

We have started by designing a non-standard interpretation for the first-order subset of the Similix specializer language shown in figure 4.5 (the full language is given in figure 2.1).

Core

As mentioned in section 2.1.7, the Similix specializer language (and thus the language of the generating extensions) is an untyped, call-by-value, functional language corresponding to a subset of Scheme with a notion of global structures and extra primitives for handling syntax trees. The particular set of primitives defined for the specializer defines two global structures, containing the list of specialization points that have already been encountered (the "seenb4" list) and the resulting specialized procedures (the "out" list). This is reflected in the semantics by defining the *Store* domain to be $\mathit{Val} \times \mathit{Val}$.

Otherwise, most of the domains are the ordinary, expected, ones. 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 simply a value and a store, that is, three values.

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

$v \in \mathit{Val}$
$v^*, [v_1, \ldots, v_n] \in \mathit{List}$
$\rho \in \mathit{Env} = \mathrm{Id} \rightarrow \mathit{Val}$
$\varphi \in \mathit{Penv}$
$f \in \mathit{Proc} = \mathit{Val}^* \rightarrow \mathit{Store} \rightarrow \mathit{Result}$
$\sigma \in \mathit{Store} = \mathit{Val} \times \mathit{Val}$
$(v, \sigma) \in \mathit{Result} = \mathit{Val} \times \mathit{Store}$
$\pi \in \mathit{Proc\text{-}Template} = \mathit{Penv} \rightarrow \mathit{Proc}$
$\theta \in \mathit{Thunk} = \mathit{Store} \rightarrow \mathit{Result}$

Figure 4.6: **Semantic Domains**

The valuation functions of the core semantics is given in figure 4.7.

Except for the primitives, much of the semantics is defined in the core of our factorization. The notable exceptions are the denotation of conditionals, that are defined using the combinator **cond**, and the denotation of procedure definitions, that are given in terms of the combinator **mk-penv**. Looking up procedure names in the resulting procedure environment is abstracted in the combinator **lookup-proc**.

Conditionals need to be specified in the interpretations for the normal abstract interpretation reasons: since the test usually cannot be resolved, we need to merge the results of the branches in some safe way.

The denotations of procedure definitions and procedure environments are abstracted so that we can create non-terminals and new grammar rules in the grammar-generating interpretation and so that we can avoid looping in the grammar-generating interpretation by defining procedures to be constant procedures (that simply return the abstract result of application) inside their own bodies. This will be explained in greater detail in section 4.2.2.

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

Standard Interpretation

The standard interpretation of the domains is shown in figure 4.9. In the standard interpretation, the domain *Penv* is simply a domain of functions from procedure names to procedures.

The standard interpretation of 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* and *Res-pgm*), since Similix has primitives for handling such structures.

Thus for example we have a Similix primitive named `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 `(primop add1 (cst 1))`, which is an element of *Res-Expr*.

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 4.8 gives the BNF corresponding to the Similix residual syntax constructors. *Res-pgm* is the similar domain of programs.

The standard interpretation of the combinators is shown in figure 4.10.

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 **mk-penv** 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

$\mathcal{P}$: Pgm → Proc-Name → *Val** → *Val*
$\mathcal{P}[\![((\text{define } (\text{F}_1\ \text{I}_1\ \ldots\ \text{I}_{n_1})\ \text{E}_1)\ \ldots\ (\text{define } (\text{F}_m\ \text{I}_1\ \ldots\ \text{I}_{n_m})\ \text{E}_m))]\!]\,\text{F}\,v^* =$
let $\varphi = \textbf{mk-penv}\,[\text{F}_1,\ldots,\text{F}_m]\,[\lambda\,\varphi\,[v_1,\ldots,v_{n_1}]\,\sigma.\,\mathcal{E}[\![\text{E}_1]\!][\text{I}_1 \mapsto v_1,\ldots,\text{I}_{n_1} \mapsto v_{n_1}](\lambda\,\text{I}.\,\mathit{undef})\,\varphi\,\sigma,\ldots,$
$\lambda\,\varphi\,[v_1,\ldots,v_{n_m}]\,\sigma.\,\mathcal{E}[\![\text{E}_m]\!][\text{I}_1 \mapsto v_1,\ldots,\text{I}_{n_m} \mapsto v_{n_m}](\lambda\,\text{I}.\,\mathit{undef})\,\varphi\,\sigma]$
in $((\textbf{lookup-proc}\,\text{F}\,\varphi\,v^*\sigma_{\mathit{init}}) \downarrow 1)$

$\mathcal{E}$: Expr → *Env* → *Penv* → *Store* → *Result*
$\mathcal{E}[\![\text{C}]\!]\,\rho\,\varphi\,\sigma = (\mathcal{C}[\![\text{C}]\!],\sigma)$
$\mathcal{E}[\![\text{I}]\!]\,\rho\,\varphi\,\sigma = (\rho\,\text{I},\sigma)$
$\mathcal{E}[\![(\text{if E}_0\ \text{E}_1\ \text{E}_2)]\!]\,\rho\,\varphi\,\sigma = \textbf{cond}(\mathcal{E}[\![\text{E}_0]\!]\,\rho\,\varphi)\,(\mathcal{E}[\![\text{E}_1]\!]\,\rho\,\varphi)\,(\mathcal{E}[\![\text{E}_2]\!]\,\rho\,\varphi)\,\sigma$
$\mathcal{E}[\![(\text{O E}_1\ \ldots\ \text{E}_n)]\!]\,\rho\,\varphi\,\sigma =$
let $(v_1,\sigma_1) = (\mathcal{E}[\![\text{E}_1]\!]\,\rho\,\varphi\,\sigma)$ *in* $\ldots$ *let* $(v_n,\sigma_n) = (\mathcal{E}[\![\text{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1})$ *in* $\mathcal{O}[\![\text{O}]\!]\,[v_1,\ldots,v_n]\,\sigma_n$
$\mathcal{E}[\![(\text{F E}_1\ \ldots\ \text{E}_n)]\!]\,\rho\,\varphi\,\sigma =$
let $f = (\textbf{lookup-proc}\,\text{F}\,\varphi)$
in let $(v_1,\sigma_1) = (\mathcal{E}[\![\text{E}_1]\!]\,\rho\,\varphi\,\sigma)$ *in* $\ldots$ *let* $(v_n,\sigma_n) = (\mathcal{E}[\![\text{E}_n]\!]\,\rho\,\varphi\,\sigma_{n-1})$ *in* $f\,[v_1,\ldots,v_n]\,\sigma_n$
$\mathcal{E}[\![(\text{let } ((\text{I E}_1))\ \text{E}_2)]\!]\,\rho\,\varphi\,\sigma = \mathit{let}\ (v,\sigma') = \mathcal{E}[\![\text{E}_1]\!]\,\rho\,\varphi\,\sigma\ \mathit{in}\ \mathcal{E}[\![\text{E}_2]\!]\,[\text{I} \mapsto v]\rho\,\varphi\,\sigma'$

$\mathcal{O}$: Op → *Val** → *Store* → *Result*
$\mathcal{O}[\![\text{cons}]\!]\,[v_1,v_2]\,\sigma = (\textbf{cons}\,v_1\,v_2,\,\sigma)$
$\mathcal{O}[\![\text{car}]\!]\,[v]\,\sigma = (\textbf{car}\,v,\sigma)$
$\ldots$
$\mathcal{O}[\![\text{generate-procedure-name!}]\!]\,[v_1,v_2]\,\sigma = \textbf{gen-proc-name}\,v_1\,v_2\,\sigma$

where
mk-penv : Proc-Name* → *Proc-Template** → *Penv*
cond : *Thunk* → *Thunk* → *Thunk* → *Thunk*
lookup-proc : Proc-Name → *Penv* → *Proc*
cons : *Val* → *Val* → *Val*
car : *Val* → *Val*
gen-proc-name : *Val* → *Val* → *Store* → *Result*

Figure 4.7: Valuation Functions

where all the procedures are mutually recursively defined.

`_sim-generate-proc-name!` is the name of one of the dedicated primitives used in the Similix specializer. It is interpreted by the combinator **gen-proc-name**. When a procedure application is to be generated by the specializer in the standard interpretation, `_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.

Note that **gen-proc-name** actually uses a single-threaded counter in the store to generate a new, unique,

RPgm	::=	add-residual-definition RDef RPgm | ϵ
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^* | abs I^* RBody | app RExpr RExpr^*

Figure 4.8: A BNF corresponding to the residual syntax constructors of Similix

$\varphi \in \mathit{Penv} = \text{Proc-Name} \to \mathit{Proc}$
$v \in \mathit{Val} = (\mathit{Nat}_\perp \oplus \mathit{Bool}_\perp \oplus \mathit{Str}_\perp \oplus \mathit{List}_\perp \oplus \mathit{Res\text{-}Expr}_\perp \oplus \mathit{Res\text{-}pgm}_\perp)$
$v^*, [v_1, \ldots, v_n] \in \mathit{List} = \bigcup \mathit{Val}^n$

Figure 4.9: Standard interpretation, domains

$\mathbf{mk\text{-}penv}_{std} = \lambda\, [\mathrm{F}_1, \ldots, \mathrm{F}_n]\, [\pi_1, \ldots, \pi_n].\ \mathit{fix}\, \lambda\, \varphi.\ [\mathrm{F}_1 \mapsto \pi_1\, \varphi, \ldots, \mathrm{F}_n \mapsto \pi_n\, \varphi](\lambda\, \mathrm{F}.\ \mathit{undef}_{Proc})$
$\mathbf{cond}_{std} = \lambda\, \theta_0\, \theta_1\, \theta_2\, \sigma.\ \mathit{let}\ (\mathit{Bool}(b), \sigma_0) = (\theta_0\, \sigma)\ \mathit{in}\ (b \to \theta_1\, \sigma_0 \,[\!]\, \theta_2\, \sigma_0)$
$\mathbf{lookup\text{-}proc}_{std} = \lambda\, \mathrm{F}\, \varphi.\ (\varphi\, \mathrm{F})$
$\mathbf{cons}_{std} = \lambda\, v\, v'.\ \mathit{let}\ \mathit{List}[w_1, \ldots, w_n] = v'\ \mathit{in}\ \mathit{inList}[v, w_1, \ldots, w_n]$
$\mathbf{car}_{std} = \lambda\, v.\ \mathit{let}\ \mathit{List}[v_1, \ldots, v_n] = v\ \mathit{in}\ v_1$
$\mathbf{gen\text{-}proc\text{-}name}_{std} = \lambda\, v_1\, v_2\, (\sigma_1, \sigma_2).\ \mathit{cases}\ (\mathit{seenb4?}\, v_1\, v_2\, \sigma_1)\ \mathit{of}$
$\mathit{isEntry}(v_n) \to (\mathit{inList}[v_n, \mathrm{True}]), (\sigma_1, \sigma_2))$
$[\!]\ \mathit{isUnit}() \to \mathit{let}\ v_n = \mathbf{gen\text{-}new\text{-}name}\, v_1\, v_2\, \sigma_1\ \mathit{in}$
$(\mathit{inList}[v_n, \mathrm{False}], ([v_1, v_2, v_n] :: \sigma_1, \sigma_2))$

Figure 4.10: Standard interpretation of the combinators

name. For simplicity we have totally omitted this from the interpretations, and we simply assume that we can, by magic, produce new, unique, strings to use as names.

Grammar-generating interpretation

In the grammar-generating interpretation, values are abstracted as pairs of right hand sides and grammars, as outlined in section 4.2.1. A right hand side is a list of alternates. An alternate is a string of terminals and non-terminals. However, as in the standard interpretation, the language imposes some additional restrictions on how we can compose these, since we intend the terminals to represent the syntax constructed by Similix.

If we consider for example the primitive `build-primop-1`, then the abstraction of this primitive takes two abstract arguments, that are both a list of alternates and a grammar. It builds a list of alternates obtained by building the applications of all the first alternates to all the second. It then merges the grammars and returns the abstract result. 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.

So as before, the domains of residual expressions and residual definitions correspond to the terms generated by the Similix syntax constructors (corresponding to the abstract syntax in figure 4.8). But now we also have to add the elements of the domain of non-terminals to the possible syntactic forms, since any syntactic form may be the result of a procedure and thus be represented by a non-terminal.

The domain of alternates also has to represent the other summands in the domain of values, as well as the residual expressions. For each of the base-value domains in the standard interpretation we add a top element (corresponding to an unknown or overspecified element that is definitely in that domain) as well as an extra top element, corresponding to elements that we know nothing about.

Finally the domain of alternates contains the non-terminals, both those representing source procedures and those representing residual procedures.

Except for the explicit top and bottom elements mentioned above, the domain of alternates is ordered discretely.

The domain of grammars is ordered by subset inclusion on the generated languages.

The grammar-generating interpretation 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*. The constant procedure environment is used to represent the "dynamically scoped" part of the procedure environment in the grammar generating interpretation.

The domain *Template* is used to represent the "statically scoped" part of the procedure environment. It is

$\varphi \in \mathit{Penv} = \mathit{Template} \times \mathit{CEnv}$
$\tau \in \mathit{Template} = \text{Proc-Name} \to \mathit{CEnv} \to \mathit{Proc}$
$\varphi_c \in \mathit{CEnv} = \text{Proc-Name} \to \mathit{CProc}$
$f \in \mathit{CProc} = (\{\lambda\, v^*\, \sigma.\, r \mid r \in \mathit{Result}\} \cup \{\mathit{undef}_{\mathit{CProc}}\})_\perp$
$v, (\alpha, \gamma) \in \mathit{Val} = \mathit{Rhs} \times \mathit{Grammar}$
$\gamma \in \mathit{Grammar} = \bigcup(\mathit{Non\text{-}Terminal} \times \mathit{Rhs})^n$
$\alpha, [a_1, \ldots, a_n] \in \mathit{Rhs} = (\bigcup \mathit{Alternate}^n)_\perp$
$a \in \mathit{Alternate} = (\mathit{Nat}_\perp^\top \oplus \mathit{Bool}_\perp^\top \oplus \mathit{Str}_\perp^\top \oplus \mathit{List}_\perp^\top \oplus \mathit{Res\text{-}Expr}_\gamma \oplus \mathit{Res\text{-}pgm}_\gamma \oplus \mathit{Non\text{-}Term\text{-}Proc}_\perp \oplus \mathit{Non\text{-}Term\text{-}Resid}_\perp)^\top$
$n \in \mathit{Non\text{-}Term\text{-}Proc} = \mathit{Str}$
$n \in \mathit{Non\text{-}Term\text{-}Resid} = \mathit{Str}$

Figure 4.11: Grammar-generating interpretation, domains

interpreted as Proc-Name $\to$ *CEnv* $\to$ *Proc*. The intention of this is that a template is a procedure environment with procedures that do not yet know which of the other procedures are constants. Given a constant procedure environment, however, it can return an actual procedure.

The domain *Penv* is interpreted as *Template* $\times$ *CEnv* and ordered with the usual pairwise ordering. The grammar generating interpretation of the combinator **lookup-proc** will look 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* part. In the *Template* 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 interpretation of domains in the grammar-generating interpretation is given in figure 4.11.

$$
\begin{array}{l}
\mathbf{mk\text{-}penv}_\gamma = \lambda\, [\mathrm{F}_1, \ldots, \mathrm{F}_n]\, [\pi_1, \ldots, \pi_n]. \\
\qquad (\mathit{fix}\ \lambda\, \tau.\ [\mathrm{F}_1 \mapsto \mathit{mk\text{-}proc}\, \mathrm{F}_1\, \pi_1\, \tau, \ldots, \\
\qquad\qquad \mathrm{F}_n \mapsto \mathit{mk\text{-}proc}\ F_n\, \pi_n\, \tau](\lambda\, \mathrm{F}.\ \mathit{undef}_{\mathit{Proc}}), (\lambda\, \mathrm{F}.\ \mathit{undef}_{\mathit{CProc}})) \\
\text{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 \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ \mathit{let}\ r_\gamma = \mathit{fix}\, \lambda\, r_\gamma.\ \pi\ (\tau, ([\mathrm{F} \mapsto \lambda\, v^* \sigma.\ (\mathit{abstract\text{-}result}\, \alpha'\, r_\gamma)]\varphi_c)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{generalize\text{-}args}\ v^*)\ (\mathit{generalize\text{-}store}\ \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{in}\ (\mathit{abstract\text{-}result}\, \alpha'\, r_\gamma) \\
\text{where } \mathit{abstract\text{-}result} = \lambda\, [n_0, n_1, n_2]\, ((\alpha_0, \gamma_0), ((\alpha_1, \gamma_1), (\alpha_2, \gamma_2))). \\
\qquad\qquad\qquad\qquad\qquad\qquad ((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))) \\
\mathbf{cond}_\gamma = \lambda\, \theta_0\, \theta_1\, \theta_2\, \sigma.\ \mathit{let}\ (([a_1, \ldots, a_n], \gamma), \sigma_0) = (\theta_0\, \sigma) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{in}\ ([a_1, \ldots, a_n] = [\mathbf{True}]) \to (\theta_1\, \sigma_0) \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\!] ([a_1, \ldots, a_n] = [\mathbf{False}]) \to (\theta_2\, \sigma_0)\ [\!]\ \mathit{merge}(\theta_1\, \sigma_0)\, (\theta_2\, \sigma_0) \\
\text{where } \mathit{merge} = \lambda\, (\alpha, \gamma)\, (\alpha', \gamma').\ (\alpha \cup \alpha', \gamma \cup \gamma') \\
\mathbf{lookup\text{-}proc}_\gamma = \lambda\, F\, (\tau, \varphi_c).\ ((\varphi_c\, \mathrm{F}) = \mathit{undef}_{\mathit{CProc}}) \to (\tau\, F\, \varphi_c)\ [\!]\ \varphi_c\, \mathrm{F} \\
\mathbf{cons}_\gamma = \lambda\, (\alpha, \gamma)\, (\alpha', \gamma').\ ([\mathbf{Cst}], \gamma \cup \gamma') \\
\mathbf{car}_\gamma = \lambda\, ([a_1, \ldots, a_n], \gamma).\ ([\mathbf{Cst}], \gamma) \\
\mathbf{gen\text{-}proc\text{-}name}_\gamma = \\
\lambda\, (\alpha, \gamma)\, (\alpha', \gamma')\, ((\alpha_1, \gamma_1), \sigma_2). \\
\quad \mathit{cases}\ (\mathit{seenb4?}\, (\alpha, \gamma)\, (\alpha', \gamma')\, (\alpha_1, \gamma_1))\ \mathit{of} \\
\qquad \mathit{isEntry}(\alpha_n) \to (([\alpha_n, [\mathbf{Bool}]]), \gamma \cup \gamma'), ((\alpha_1, \gamma_1), \sigma_2)) \\
\qquad [\!]\ \mathit{isUnit}() \to \mathit{let}\ \alpha_n = \mathbf{gen\text{-}new\text{-}name}_\gamma\, (\alpha, \gamma)\, (\alpha', \gamma')\, (\alpha_1, \gamma_1)\ \mathit{in} \\
\qquad\qquad (([\alpha_n, [\mathbf{False}]], \gamma \cup \gamma'), ([\alpha, \alpha', \alpha_n] :: \alpha_1, (\gamma \cup \gamma' \cup \gamma_1)), \sigma_2)
\end{array}
$$

Figure 4.12: Grammar-generating interpretation of the combinators

Figure 4.12 gives the grammar-generating interpretation of the combinators.

Some of these follow the usual pattern of abstract interpretation: the non-standard interpretation of the combinator **cond** merges 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 combinator **mk-penv** takes a list of procedure names and a list of procedure templates and returns a procedure environment. The procedure environment consists of a template part and a constant procedure environment part. In the template part, the procedure name is bound to a function that will take a constant procedure environment to a procedure. The procedure is again a function that takes a list of abstract values and an abstract store. The result of this function is the fixed point of the application of the procedure template to 1) the procedure environment built from the template and the constant procedure environment extended with the procedure name bound to the the constant procedure returning the fixed point 2) the list of values 3) the store.

The intention of this 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 interpretation has a fixed point over the denotation of the procedure, the non-standard interpretation has a fixed point over the denotation of the *body* of the procedure.

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

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 interpretation of **gen-proc-name** can return the flag **False** if it does not find the abstract values in the abstract store, but only the flag **Bool** if it does find them.

4.2.3 Safety of the analysis

We shall prove safety of the analysis using logical relations [Jones & Nielson 91].

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

We denote the language generated by a grammar γ by $L(\gamma)$. By an abuse of notation, we will use $L(\alpha, \gamma)$ to denote the language generated by a term α, where the non-terminals in α may be defined in γ and if they are not, they are taken to mean $\bot$, except for the predefined non-terminal **Cst**, which is taken to denote the set of constants, and other similar predefined non-terminals that we use for convenience. We order the set of languages generated by L by subset inclusion, with the empty language, generated by the empty grammar, as $\bot$. For convenience we define $L(\alpha, \gamma)$ to be downwards closed, that is, if $v_1 \in L(\alpha, \gamma)$ and $v_0 \sqsubseteq v_1$, then $v_0 \in L(\alpha, \gamma)$. We note that L is monotonous.

The relations between compound domains are given in figure 4.13. We also need a relation $P_{Template}$ between standard procedure environments and the domain of templates. The relations are built up from the relations between their components in the usual way, except P_{Proc} and $P_{Template}$. It would not be possible to compare the elements of *Template* in the usual way, since there is no corresponding domain in the standard interpretation.

Congruence between the augmented standard and the grammar-generating interpretations

To prove that the grammar generating interpretation is congruent with the standard interpretation, 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 $P_{\textit{Proc-Name}^* \to \textit{Proc-Template}^* \to \textit{Penv}}(\textbf{mk-penv}_{std}, \textbf{mk-penv}_{\gamma})$ holds, *i.e.*, that 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 $\textit{Proc-Template}^*_{std} \times \textit{Proc-Template}^*_{\gamma}$ we have $P_{\textit{Proc-Template}}(\pi_i, \pi_{i\gamma})$ for $i = 1, \ldots, n$ implies $P_{\textit{Penv}}(\textbf{mk-penv}_{std}[\mathrm{F}_1, \ldots, \mathrm{F}_n]\,[\pi_1, \ldots, \pi_n], \textbf{mk-penv}_{\gamma}[\mathrm{F}_1, \ldots, \mathrm{F}_n]\,[\pi_{1\gamma}, \ldots, \pi_{n\gamma}])$.

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

Figure 4.13: Relations between the standard and the abstract domains

Since we need to use fixed point induction, we need the relation we want to establish to be an inclusive predicate over the two domains. Most of the predicates are trivially inclusive, as outlined:
P_{Val}: since Val_{std} is flat and L is monotonous, this is inclusive.
P_{Store}: Since *Store* is $Val \times Val$, inclusivity of P_{Store} follows from inclusivity of P_{Val}.
P_{Result}: Inclusivity of P_{Result} follows from inclusivity of P_{Val} and P_{Store}.
P_{Proc}: Inclusivity of P_{Proc} clearly follows from inclusivity of P_{Val}.
P_{Penv}: Inclusivity of P_{Penv} clearly follows from inclusivity of P_{Proc}.

The predicate $P_{Template}$, that we use in the proof for **mk-penv**, however, is not immediately inclusive, because of the use of implication.

Theorem 2 *The predicate $P_{Template}$ over $Penv_{std} \times Template$ is inclusive.*

Proof: To show that $P_{Template}$ is inclusive, we consider an arbitrary chain (φ_i,τ_i) in $Penv_{std} \times Template$, for which $P_{Template}(\varphi_i,\tau_i)$ holds for all i and we show that $P_{Template}(\bigsqcup(\varphi_i),\bigsqcup(\tau_i))$ holds.

We need to show $Q(\bigsqcup(\varphi_i),\varphi_c) \Rightarrow P_{Penv}(\bigsqcup(\varphi_i),(\bigsqcup(\tau_i),\varphi_c))$ for all $\varphi_c \in CEnv$, so we consider an arbitrary φ_c. Let $Q_{\varphi_c}(\varphi)$ be $\forall \mathrm{F} \in \text{Proc-Name} : \varphi_c\,\mathrm{F} = undef_{CProc} \vee P_{Proc}(\varphi\,\mathrm{F},\varphi_c\,\mathrm{F})$. Then we have two cases:

- $Q_{\varphi_c}(\bigsqcup(\varphi_i))$ is false. In this case the implication is trivially true and we are done.
- $Q_{\varphi_c}(\bigsqcup(\varphi_i))$ is true. In this case we consider two subcases for each F:
  - $\varphi_c\,\mathrm{F} = undef_{CProc}$. In this subcase the disjunction must also be true for all φ_i.
  - $\varphi_c\,\mathrm{F} \neq undef_{CProc}$. In this subcase $P_{Proc}(\sqcup\varphi_i F, \varphi_c F)$ holds, so by lemma 1, stated below, the disjunction is true for all φ_i.

Thus we have that if $Q_{\varphi_c}(\bigsqcup(\varphi_i))$ is true, then $Q_{\varphi_c}(\varphi_i)$ is true for all i. Since $P_{Template}(\varphi_i,\tau_i)$ holds for all i, this implies that $P_{Penv}(\varphi_i,(\tau_i,\varphi_c))$ holds for all i so by inclusivity of P_{Penv}, we get that $P_{Penv}(\bigsqcup(\varphi_i),(\bigsqcup(\tau_i),\varphi_c))$ holds for φ_c. Since φ_c was arbitrary, we have that $P_{Template}(\bigsqcup(\varphi_i),\bigsqcup(\tau_i))$. □

Lemma 1 *For a chain (φ_i) in $Penv_{std}$, an f_γ in $Proc_\gamma$ and a procedure name F in* Proc-Name, *$P_{Proc}(\bigsqcup(\varphi_i)\,\mathrm{F}, f_\gamma)$ implies $\forall i : P_{Proc}(\varphi_i\,\mathrm{F}, f_\gamma)$.*

Proof: For an arbitrary i, consider, for arbitrary v^* and σ, $\varphi_i\,\mathrm{F}\,v^*\,\sigma$. We know that for any v^*_γ and σ_γ: $P_{Result}(\bigsqcup(\varphi_i)\,\mathrm{F}\,v^*\,\sigma, f_\gamma\,v^*_\gamma\,\sigma_\gamma)$. Since $\varphi_i \sqsubseteq_{Penv} \bigsqcup(\varphi_i)$ we have $\varphi_i\,\mathrm{F} \sqsubseteq_{Proc} (\bigsqcup(\varphi_i))\,\mathrm{F}$, so $\varphi_i\,\mathrm{F}\,v^*\,\sigma \sqsubseteq_{Result} (\bigsqcup(\varphi_i))\,\mathrm{F}\,v^*\,\sigma$. If we let $[v^1_i,[v^2_i,v^3_i]] = \varphi_i\,\mathrm{F}\,v^*\,\sigma$ and $[v^1,[v^2,v^3]] = (\bigsqcup(\varphi_i))\,\mathrm{F}\,v^*\,\sigma$, this means that $v^j_i \sqsubseteq v^j, j = 1,2,3$. So since any set generated by L is downwards closed $\varphi_i\,\mathrm{F}\,v^*\,\sigma \in L(f_\gamma\,v^*_\gamma\,\sigma_\gamma)$. Since v^* and σ were arbitrary, we have $P_{Proc}(\varphi_i\,\mathrm{F}, f_\gamma)$. Since i was arbitrary, we have $\forall i : P_{Proc}(\varphi_i\,\mathrm{F}, f_\gamma)$. □

The following lemma is needed in the proof for the combinator **mk-penv**.

Lemma 2 *Assume we are given* F *in* Proc-Name, *π_γ in Proc-Template$_\gamma$, τ in Template, φ_c in CEnv, v^*_γ in Val^*_γ, and σ_γ in Store$_\gamma$. Then for $i = 1,2,3$:*

$$L((\textit{mk-proc}\,\mathrm{F}\,\pi_\gamma\,\tau\,\varphi_c\,v^*_\gamma\,\sigma_\gamma) \downarrow i) = L((\textit{fix}\ \lambda\, r.\ \pi_\gamma\,(\tau, [\mathrm{F} \mapsto \lambda\, v^*_\gamma\,\sigma_\gamma.\ (\textit{abstract-result}\,\alpha\,r)]\varphi_c)\,v^*_\gamma\,\sigma_\gamma) \downarrow i)$$

where α is a sequence of three fresh non-terminals.

Proof: If we have a value (α', γ) and n is a non-terminal not used in α' or in γ, then clearly $L(\alpha', \gamma)$ is equal to $L(n, \{n, \alpha'\} \cup \gamma)$. Since *mk-non-terminal* generates fresh non-terminals, the lemma then follows directly from the definition of *mk-proc*. □

Lemma 3 *For the combinator* **mk-penv** *we have that*
$P_{\mathit{Proc\text{-}Name}^* \to \mathit{Proc\text{-}Template}^* \to \mathit{Penv}}(\mathbf{mk\text{-}penv}_{std}, \mathbf{mk\text{-}penv}_{\gamma})$
holds.

Proof: We assume we are given $[\mathrm{F}_1, \ldots, \mathrm{F}_n] \in \text{Proc-Name}^*$ 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 $i = 1, \ldots, n$ $P_{\mathit{Proc\text{-}Template}}(\pi_i, \pi_{i\gamma})$ holds.

By lemma 4 below, we have that P_{Template} holds between $\mathit{fix}\ \lambda\, \varphi.\ [\mathrm{F}_1 \mapsto \pi_1\, \varphi, \ldots, \mathrm{F}_n \mapsto \pi_n\, \varphi](\lambda\, \mathrm{F}.\ \mathit{undef}_{\mathit{Proc}})$ and $\mathit{fix}\ \lambda\, \tau.\ [\mathrm{F}_1 \mapsto \mathit{mk\text{-}proc}\, \mathrm{F}_1\, \pi_1\, \tau, \ldots, \mathrm{F}_n \mapsto \mathit{mk\text{-}proc}\, F_n\, \pi_n\, \tau](\lambda\, \mathrm{F}\, \varphi_c.\ \mathit{undef}_{\mathit{Proc}})$, so since we clearly have that $\lambda\, \mathrm{F}.\ \mathit{undef}_{\mathit{CProc}}$ fulfills condition Q,
$P_{\mathit{Penv}}(\mathbf{mk\text{-}penv}_{std}[\mathrm{F}_1, \ldots, \mathrm{F}_n]\, [\pi_1, \ldots, \pi_n], \mathbf{mk\text{-}penv}_{\gamma}[\mathrm{F}_1, \ldots, \mathrm{F}_n]\, [\pi_{1\gamma}, \ldots, \pi_{n\gamma}])$ follows. □

Lemma 4 *Let* $[\mathrm{F}_1, \ldots, \mathrm{F}_n]$ *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* $i = 1, \ldots, n$ $P_{\mathit{Proc\text{-}Template}}(\pi_i, \pi_{i\gamma})$ *holds.*

Let $H = \lambda\, \varphi.\ [\mathrm{F}_1 \mapsto \pi_1\, \varphi, \ldots, \mathrm{F}_n \mapsto \pi_n\, \varphi](\lambda\ \mathrm{F}.\ \mathit{undef}_{\mathit{Proc}})$ *and let* $H_{\gamma} = \lambda\ \tau.\ [\mathrm{F}_1 \mapsto \mathit{mk\text{-}proc}\, \mathrm{F}_1\, \pi_1\, \tau, \ldots, \mathrm{F}_n \mapsto \mathit{mk\text{-}proc}\, F_n\, \pi_n\, \tau](\lambda\, \mathrm{F}\, \varphi_c.\ \mathit{undef}_{\mathit{Proc}})$.

Then we have that $P_{\mathit{Template}}(\mathit{fix}\ H, \mathit{fix}\ H_{\gamma})$ *holds.*

Proof:

Let us abbreviate $H^i(\bot)$ by φ^i and $H^i_{\gamma}(\bot)$ by τ^i.

We use fixed point induction:
Base case: $P_{\mathit{Template}}(\bot, \bot)$ holds, since the value $\bot$ is in all languages generated by L.
Step: Assume that for all $j \leq i$, $P_{\mathit{Template}}(\varphi^j, \tau^j)$ holds. Show that $P_{\mathit{Template}}(\varphi^{i+1}, \tau^{i+1})$ holds.

That is, we have to show that for all φ_c so that $Q(\varphi^{i+1}, \varphi_c)$ holds, we have that $P_{\mathit{Penv}}(\varphi^{i+1}, (\tau^{i+1}, \varphi_c))$ holds.

So we assume that we are given a φ_c, so that $Q(\varphi^{i+1}, \varphi_c)$ holds.

Then we want to prove for all F_j in $[\mathrm{F}_1, \ldots, \mathrm{F}_n]$ that

$$P_{\mathit{Proc}}(\mathbf{lookup\text{-}proc}_{std}\, \mathrm{F}_j\, \varphi^{i+1}, \mathbf{lookup\text{-}proc}_{\gamma}\, \mathrm{F}_j\, (\tau^{i+1}, \varphi_c))$$

holds (since φ^{i+1} is undefined for all other F in Proc-Name). This falls in two parts.

- $\varphi_c\, \mathrm{F}_j \neq \mathit{undef}_{\mathit{CProc}}$: In this case $\mathbf{lookup\text{-}proc}_{\gamma}\, \mathrm{F}_j\, (\tau^{i+1}, \varphi_c)$ is equal to $\varphi_c\, \mathrm{F}_j$, so by the assumption on φ_c, we are done.
- $\varphi_c\, \mathrm{F}_j = \mathit{undef}_{\mathit{CProc}}$: In this case we find that

  $$\mathbf{lookup\text{-}proc}_{std}\, \mathrm{F}_j\, \varphi^{i+1} = \pi_j\, \varphi^i \text{ and } \mathbf{lookup\text{-}proc}_{\gamma}\, \mathrm{F}_j\, (\tau^{i+1}, \varphi_c) = \mathit{mk\text{-}proc}\, \mathrm{F}_j\, \pi_{j\gamma}\, \tau^i\, \varphi_c$$

  So we need to show that for all (v^*, v^*_{γ}) in $\mathit{Val}^* \times \mathit{Val}^*_{\gamma}$ and $(\sigma, \sigma_{\gamma})$ in $\mathit{Store} \times \mathit{Store}_{\gamma}$, $P_{\mathit{Result}}(\pi_j\, \varphi^i\, v^*\, \sigma, \mathit{mk\text{-}proc}\, \mathrm{F}_j\, \pi_{j\gamma}\, \tau^i\, \varphi_c\, v^*_{\gamma}\, \sigma_{\gamma})$ holds. By lemma 2 this is equivalent to showing that

  $$P_{\mathit{Result}}(\pi_j\, \varphi^i\, v^*\, \sigma, \mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*_{\gamma}\, \sigma_{\gamma}.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_{\gamma}\, \sigma_{\gamma})$$

  holds, where α is a sequence of three fresh non-terminals.

  Now we unfold the fixed point once:
  $\mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_{\gamma}\, \sigma_{\gamma}) =$
  $\pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, (\mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_{\gamma}\, \sigma_{\gamma}))]\varphi_c)\, v^*_{\gamma}\, \sigma_{\gamma}$
  so we can get that P_{Result} holds if we can prove that
  $P_{\mathit{Penv}}(\varphi^i,$
  $(\tau^i, [\mathrm{F}_j \mapsto$

$$\lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, (\mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_\gamma\, \sigma_\gamma))]\varphi_c))$$

holds.

Let us abbreviate
$\lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, (\mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_\gamma\, \sigma_\gamma))$
by $f_{\gamma j}$.

Since we have $P_{\mathit{Template}}(\varphi^i, \tau^i)$, P_{Penv} will follow if we can prove $Q(\varphi^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$.

Since for all F in Proc-Name, we have that $\varphi^i \mathrm{F} \sqsubseteq \varphi^{i+1}\mathrm{F}$, it follows that $P_{\mathit{Proc}}(\varphi^{i+1}\mathrm{F}, \varphi_c\, \mathrm{F})$ implies $P_{\mathit{Proc}}(\varphi^i \mathrm{F}, \varphi_c\, \mathrm{F})$. So all we need to prove is $P_{\mathit{Proc}}(\varphi^i \mathrm{F}_j, f_{\gamma j})$.

Unfortunately this is not so simple, so we proceed by proving $Q(\varphi^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$ by induction from 0 to i.

k = 0: Clearly $Q(\varphi^0, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$ holds.

Assume $Q(\varphi^k, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$ holds and $k < i$. Show $Q(\varphi^{k+1}, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$.

As mentioned above, this is trivial for $\mathrm{F} \neq \mathrm{F}_j$. For $\mathrm{F} = \mathrm{F}_j$ we need to prove for all (v^*, w^*_γ) and (σ, σ'_γ) that

$$P_{\mathit{Result}}(\pi_j\, \varphi^k\, v^*\, \sigma, f_{\gamma j}\, w^*_\gamma\, \sigma'_\gamma)$$

holds, so again we use lemma 2 and unfold the fix in $f_{\gamma j}$, giving
$f_{\gamma j}\, w^*_\gamma\, \sigma'_\gamma =$
$\pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, (\mathit{fix}\ \lambda\, r_j.\ \pi_{j\gamma}\, (\tau^i, [\mathrm{F}_j \mapsto \lambda\, v^*\, \sigma.\ (\mathit{abstract\text{-}result}\, \alpha\, r_j)]\varphi_c)\, v^*_\gamma\, \sigma_\gamma))]\varphi_c)\, v^*_\gamma\, \sigma_\gamma$
so again we get that P_{Result} holds if $P_{\mathit{Penv}}(\varphi^k, (\tau^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c))$ holds. Now by lemma 5 we have that $P_{\mathit{Template}}(\varphi^k, \tau^i)$ holds, since $k < i$. Since by the inductive hypothesis we get that $Q(\varphi^k, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$ holds, we conclude that $P_{\mathit{Penv}}(\varphi^k, (\tau^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c))$ holds. So we get that P_{Result} holds as needed, so we have shown $Q(\varphi^{k+1}, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$.

So we conclude $Q(\varphi^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c)$, which gives us $P_{\mathit{Penv}}(\varphi^i, (\tau^i, [\mathrm{F}_j \mapsto f_{\gamma j}]\varphi_c))$ as needed. So we conclude that $P_{\mathit{Result}}(\pi_j\, \varphi^i\, v^*\, \sigma, \mathit{mk\text{-}proc}\, \mathrm{F}_j\, \pi_{j\gamma}\, \tau^i\, \varphi_c)$ holds and we are done. □

Lemma 5 *If 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* $\mathit{Proc\text{-}Template}^*_{\mathit{std}} \times \mathit{Proc\text{-}Template}^*_\gamma$, *so that for* $i = 1, \ldots, n$ $P_{\mathit{Proc\text{-}Template}}(\pi_i, \pi_{i\gamma})$ *holds and let*

$$H = \lambda\, \varphi.\ [\mathrm{F}_1 \mapsto \pi_1\, \varphi, \ldots, \mathrm{F}_n \mapsto \pi_n\, \varphi]\lambda\, \mathrm{F}.\ \mathit{undef}_{\mathit{Proc}}$$

and

$$H_\gamma = \lambda\, \tau.\ [\mathrm{F}_1 \mapsto \mathit{mk\text{-}proc}\, \mathrm{F}_1\, \pi_{1\gamma}\, \tau, \ldots, \mathrm{F}_n \mapsto \mathit{mk\text{-}proc}\, F_n\, \pi_{n\gamma}\, \tau](\lambda\, \mathrm{F}\, \varphi_c.\ \mathit{undef}_{\mathit{Proc}})$$

then if we have for all $j \leq i$ *that* $P_{\mathit{Template}}(H^j(\bot), H^j_\gamma(\bot))$, *then for all* $m \leq n \leq i$ *we have* $P_{\mathit{Template}}(H^m(\bot), H^n_\gamma(\bot))$.

Proof: By induction over i and m, similar to the induction above. Omitted.

Lemma 6 *For the combinator* **lookup-proc** *we have that*

$$P_{\mathit{Penv} \to \mathit{Proc\text{-}Name} \to \mathit{Proc}}(\mathbf{lookup}_{\mathit{std}}\ , \mathbf{lookup\text{-}proc}_\gamma)$$

holds.

Proof: Assume we are given an F in Proc-Name and $(\varphi, \varphi_\gamma)$ in $\mathit{Penv}_{\mathit{std}} \times \mathit{Penv}_\gamma$ so that $P_{\mathit{Penv}}(\varphi, \varphi_\gamma)$. Then we need to show $P_{\mathit{Proc}}(\mathbf{lookup\text{-}proc}_{\mathit{std}}\, F\, \varphi, \mathbf{lookup\text{-}proc}_\gamma\, F\, \varphi_\gamma)$. But this follows exactly from $P_{\mathit{Penv}}(\varphi, \varphi_\gamma)$. □

Lemma 7 *For the combinator* **cond** *we have that* $P_{\mathit{Thunk} \to \mathit{Thunk} \to \mathit{Thunk} \to \mathit{Thunk}}(\mathbf{cond}_{\mathit{std}}, \mathbf{cond}_\gamma)$ *holds.*

Proof: To show this, we assume $P_{Thunk}(\theta_0, \theta_{\gamma_0}), P_{Thunk}(\theta_1, \theta_{\gamma_1}), P_{Thunk}(\theta_2, \theta_{\gamma_2})$ and $P_{Store}(\sigma, \sigma_\gamma)$. From this we need to show $P_{Result}(\mathbf{cond}_{std}\,\theta_0\,\theta_1\,\theta_2\,\sigma, \mathbf{cond}_\gamma\,\theta_{\gamma_0}\,\theta_{\gamma_1}\,\theta_{\gamma_2}\,\sigma_\gamma)$. If we unfold the definitions, this is
$P_{Result}(let\ (Bool(b), \sigma_0) = (\theta_0\,\sigma)\ in\ (b \to \theta_1\,\sigma_0 \,[\!]\, \theta_2\,\sigma_0),$
$\quad let\ (([a_1, \ldots, a_n], \gamma), \sigma_{\gamma_0}) = (\theta_{\gamma_0}\,\sigma)\ in$
$\quad\quad ([a_1, \ldots, a_n] = [\mathbf{True}]) \to (\theta_{\gamma_1}\,\sigma_{\gamma_0}) \,[\!]\, (([a_1, \ldots, a_n] = [\mathbf{False}]) \to (\theta_{\gamma_2}\,\sigma_{\gamma_0}) \,[\!]\, merge(\theta_{\gamma_1}\,\sigma_{\gamma_0})\,(\theta_{\gamma_2}\,\sigma_{\gamma_0}))).$
If $(\theta_0\,\sigma)$ is not a boolean, then the final result is $(\bot, \bot)$, which is in any language, so the predicate holds. If $(\theta_0\,\sigma)$ is $(Bool(b), \sigma_0)$, then by assumption $P_{Result}((Bool(b), \sigma_0), \theta_{\gamma_0}\,\sigma_\gamma)$. Now we have three cases:
$(\theta_{\gamma_0}\,\sigma_\gamma) = (([\mathbf{True}], \gamma), \sigma_\gamma)$: Since $P_{Val}(Bool(b), ([\mathbf{True}], \gamma))$, this implies that $b =$ TRUE, so by assumption we are done.
$(\theta_{\gamma_0}\,\sigma_\gamma) = (([\mathbf{False}], \gamma), \sigma_\gamma)$: similar.
$(\theta_{\gamma_0}\,\sigma_\gamma) = (([a_1, \ldots, a_n], \gamma), \sigma_\gamma)$: in this case $(\mathbf{cond}_\gamma\,\theta_{\gamma_0}\,\theta_{\gamma_1}\,\theta_{\gamma_2}\,\sigma_\gamma)$ is $merge(\theta_{\gamma_1}\,\sigma_\gamma)\,(\theta_{\gamma_2}\,\sigma_\gamma)$. Since $L(\alpha, \gamma) \subseteq L(merge\,(\alpha, \gamma)\,(\alpha', \gamma'))$, we have either
$(\mathbf{cond}_{std}\,\theta_0\,\theta_1\,\theta_2\,\sigma) \downarrow i = (\theta_2\,\sigma_\gamma) \downarrow i \in L((\theta_{\gamma_1}\,\sigma_\gamma) \downarrow i) \subseteq L(merge\,(\theta_{\gamma_1}\,\sigma_\gamma) \downarrow i\,(\theta_{\gamma_2}\,\sigma_\gamma) \downarrow i)$ for i = 1,2,3
or the converse. □

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 operations such as cons or add1, that will always return simple results such as **Cst** or **Num**. The proof of safety for this class of primitives relies on the language generating function L taking **Cst** to the set of all constants and so on.

The constructors, such as `build-cond`, are only slightly more complicated, relying on the proper handling of alternates as well as L.

The combinators for the management operators, such as **gen-proc-name**, **add-residual-definition** and **get-residual-program** require a bit more care, specially the name generators (there are in fact several in Similix) and of course **gen-proc-name**, that involves the "seen before" list.

4.3 Examples

Although the example given in section 4.1 serve to illustrate the technique, it is not very interesting as a program.

To see where the results of the analysis may in fact contain interesting information, we can consider the toy interpreter shown in figure 4.14. It interprets a fragment of an imperative language using an environment and a store. When we specialize the interpreter with respect to a program in the language, we expect all environment actions to disappear in the target program. This indeed happens for the sample program shown in figure 4.15, as can be seen in the target program in figure 4.16. But can we say that it is always true?

Using the analysis described above, we can derive a grammar describing the family of target programs. If this family does not contain any environment operations, we have proven the claim.

The BNF obtained (by hand) from the generating extension of the interpreter is given in figure 4.17. Clearly the language described by this BNF does not contain any environment operations or any variables derived from the environment of the interpreter.

A similar example is the interpreter for an Algol-like language presented in [Consel & Danvy 91]. By partial evaluation a compiler is generated from this interpreter and it is shown by example that the compiler performs the type-checking. Using the analysis above, it would be possible to give a formal proof that the target programs do not contain any type-checking.

4.4 Possible extensions

Except for the applications to partial evaluation, techniques like the one presented here might also be useful in connection with two-level languages. Since the use of two-level languages is not based on self-application, the approach of section 3.4 might be more useful here.

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

(define (_p p i)
    (let ((res (_d (car p) (mk-initial-env))))
        (_c (cadr p) (car res) (initial-store (cdr res) i))))

(define (_d d r-l)
    (if (null? d)
        r-l
        (_d (cdr d) (update-r (car d) r-l))))

(define (_c c r s)
    (let ((tag (car c)))
        (cond
         ((equal? tag ':=)   (let ((v (_e (caddr c) r s)))
                                (update-s (lookup-r (cadr c) r) v s)))
         ((equal? tag 'seq)  (let ((s (_c (cadr c) r s)))
                                (_c (caddr c) r s)))
         ((equal? tag 'if)    (let ((v (_e (cadr c) r s)))
                                 (if (boolean? v)
                                     (if v
                                         (_c (caddr c) r s)
                                         (_c (cadddr c) r s))
                                     s)))
         (else 'error))))

(define (_e e r s)
    (let ((tag (car e)))
        (cond
         ((equal? tag 'id)    (lookup-s (lookup-r (cadr e) r) s))
         ((equal? tag 'cst)   (cadr e))
         (else 'error))))
```

Figure 4.14: A simple interpreter for a tiny language

```
(((a)  (b)) .
 ((seq (if (id a)
           (:= b (cst 0))
           (:= b (cst 1)))
       (:= a (cst 5)))))
```

Figure 4.15: A sample program in the language interpreted by the program in figure 4.14

```
(loadt "int.adt")
(loadt "scheme.adt")
(define (_p-0 i_0)
    (let ([s_1 (initial-store 2 i_0)])
        (update-s 0
                  5
                  (let ([v_2 (lookup-s 0 s_1)])
                      (if (boolean? v_2)
                          (if v_2 (update-s 1 0 s_1) (update-s 1 1 s_1))
                          s_1)))))
```

Figure 4.16: The tiny interpreter specialized with respect to the sample expression in figure 4.15

specialize-0	::=	`(define (`ID_p IDi`) (let ((`IDs `(initial-store` **Cst** IDi`))) `expr-3`))`
expr-3	::=	`(update-s` **Cst** expr-4 IDs`)` | `(let ((`IDs expr-3`)) `expr-3`)` | `(`ID_c-0 IDs`)` | **Cst**
ID_c-0	::=	`(define (`ID_c-0 IDs`) (let ((`IDv expr-4`)) (if (boolean?` IDv`) (if` IDv expr3 expr3`) IDs)))`
expr-4	::=	`(lookup-s` **Cst** IDs`)` | **Cst**

Figure 4.17: BNF describing the possible results of specializing the tiny interpreter with the first argument static and the second argument dynamic

The analysis presented here makes a very coarse generalization on the arguments when procedures are called: the actual value is completely forgotten and the parameter is simply bound to Cst. This helps to simplify the formal specification of the analysis, but is probably not satisfactory in the long run.

There are several ways in which this could be refined.

One possibility would be to pass the abstract arguments to the procedure. In general this would cause the grammar generating interpretation to loop on all but the most simple programs. This can be solved by limiting the possible arguments to some finite set and generalizing the actual argument to its best approximation in the set.

Another, perhaps more promising, extension is to generate grammar rules for the parameters and bind the formals to the non-terminals describing that argument, as in [Jones 87]. This would greatly increase the unreadability of the resulting grammars, however, so a way of limiting this to those arguments that are actually involved in constructions seems necessary. This might possibly be done by a pre-analysis.

In order to make reliable estimates about properties such as size or time and space consumption by specialized programs, the analysis presented here is not quite strong enough.

One promising extension of the analysis, that would allow us to reason about these properties, is to use a less trivial abstraction of the static data, containing type information or a structural description in the style of partially static structures [Mogensen 88]. This structural information could then be used to attach attributes to the grammar when the generating extension tests or de-structures a known structure.

4.5 Analysis of a higher-order language

When extending a first order analysis to higher order, one is faced with the choice of abstracting the higher order concrete values either as functions from abstract values to abstract values or as a new kind of ground abstract values (such a closures).

In a language such as Scheme, there is not really any semantic difference between a top-level procedure, defined with the `define` keyword, and a higher-order value defined with `lambda` and possibly `letrec`. In a system

such as lambda-mix there is in fact only one kind of procedures. This means that it may be hard to justify a difference in the abstraction of globally defined procedures and of locally defined higher-order values. Since, in the first-order case, procedures are abstracted as functions, this seems to argue that an abstraction of higher order values as functions over abstract values would be the appropriate extension. But this method is known to lead to very slow implementations, due to the height of the involved function domains, so it might not be what we want after all.

If we try to look at the present abstraction, the main principle of the first order analysis can be phrased very simply: replace recursively defined functions by constant functions returning recursively defined values. Stated this way, there are clearly two possibilities when going to higher order:

- Maintain that the domain of values is abstracted by a domain of grammars. Since values may now be higher order, this means that the grammars must be able to represent higher order values, either by some notion of higher order grammars or by some first order representation that needs to be interpreted if the higher order value is applied.
- Maintain that grammars represent only ground values, giving us an abstract domain of values that is a direct sum of grammars and higher order values. Again the higher order values can be either functions over abstract values or some ground representation such as closures.

Abstracting higher order values as higher order grammars seems impractical, since there is nothing to stop it from looping if the higher order value is recursive.

Using first order grammars (the usual kind) that then needs to be interpreted in some way seems impractical for basically the same reason. Since we use recursion in the grammars to keep the interpretation from looping, we cannot also interpret the grammars, since they might "loop".

Abstracting higher order values as functions over grammars would correspond to the treatment of the top-level procedures, but has its own set of problems. It will be necessary to know how many arguments to feed to a function before the result is a ground value (since in the first-order case the fixed point is taken *after* the function has been applied to its arguments, that is, over a ground value). In other words, some kind of simple type information is required. Some kind of closure representation, possibly obtained by a closure analysis [Shivers 88], might be a way to obtain a safe approximation of such information in a dynamically typed language like Scheme.

Since Similix-2 is higher order, it is a definite goal to obtain an analysis that can, at least theoretically, handle higher order programs.

4.6 Implementation

We are currently working on an implementation of the simple analysis outlined in section 4.2. We plan to finish this implementation no later than January 1992, at which point the analysis will undoubtedly reveal itself to be overly simplistic. The fact that it is first order, while the partial evaluator we investigate is higher order, will probably not be the largest practical problem, since Similix-2 produces first-order generating extensions for a large, if not yet well-defined, class of programs. The very coarse generalization in the abstract parameter passing, however, will probably prove to be a severe limitation.

Since the programs we analyze are automatically generated it is impractical to analyze a significant number of programs by hand in order to distinguish the effects of different refinements in the analysis. It is the intention that the prototype implementation should help this process. Obviously some refinements will be pure extensions of the existing framework, and thus can be implemented as extensions of the prototype. Other refinements may require major reorganizations, in which case we will probably not implement them.

Chapter 5

Conclusion

This proposal describes the beginning of a project for predicting intensional properties of specialized programs.

Several approaches have been investigated and the idea of analyzing generating extensions by means of abstract interpretation has been selected.

An abstract representation of sets of residual programs as grammars have been selected, both as a good basis for abstract interpretation and as a convenient representation of programs. A simple analysis of a first order language, corresponding to a subset of the language of an actual partial evaluator, has been designed and its core parts have been proven safe, using the technique of logical relations.

An implementation of this simple analysis is currently under way and will be finished in the near future.

The simple analysis is clearly too coarse, and some refinement is required. Furthermore, an extension to higher order is desirable, since the partial evaluator we are using treats higher order programs.

We intend to work on designing and proving these extensions in the spring and summer of 1992. If this proceeds smoothly, we intend to spend the end of the summer on a more extensive implementation, that could fit directly into the Similix system. If not, there will undoubtedly be some interesting problems to relate.

The project will be finished in the spring semester of 1993, preferably the beginning.

Bibliography

[Ball 79] J. Eugene Ball: *Predicting the Effects of Optimization on a Procedure Body*, in *SIGPLAN Symposium on Compiler Construction*, Denver, Colorado, August 1979, SIGPLAN Notices, vol 14, no 8 (August 79)

[Barzdins & Bulyonkov 88] G. J. Barzdins & M. A. Bulyonkov: *Mixed Computation and Translation: Linearisation and Decomposition of Compilers.* Preprint 791 from Computing Centre of Siberian division of USSR Academy of Sciences, p.32, Novosibirsk (1988) (in Russian)

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

[Bondorf & Danvy 91] Anders Bondorf & Olivier Danvy: *Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Type*, to appear in *Science of Computer Programming* (1991)

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

[Bondorf 91b] Anders Bondorf: *Similix Manual, system version 3.0*, DIKU Report 91/9, DIKU, Department of Computer Science, University of Copenhagen (June 1991)

[Bulyonkov 84] M. A. Bulyonkov: *Polyvariant Mixed Computation for Analyzer Programs*, Acta Informatica, vol 21, fasc 5, pp 473-484 (1984)

[Consel 88] Charles Consel: *New Insights into Partial Evaluation: The Schism Experiment*, in *ESOP '88, 2nd European Symposium on Programming*, Nancy, France, March 1988, H. Ganzinger (ed.), Lecture Notes in Computer Science, vol 300, pp 236-246, Springer-Verlag (1988)

[Consel & Danvy 89] Charles Consel & Olivier Danvy: *Partial Evaluation of Pattern Matching in Strings*, Information Processing Letters, vol 30, no 1 (January 1989)

[Consel 90] Charles Consel: *Binding Time Analysis for Higher Order Untyped Functional Languages*, in proceedings of *1990 ACM Conference on Lisp and Functional Programming*, Nice, France, pp 264-272 (1990)

[Consel & Danvy 90] Charles Consel & Olivier Danvy: *From Interpreting to Compiling Binding Times*, in *ESOP '90. 3rd European Symposium on Programming*, Copenhagen, Denmark, May 1990, N. Jones (ed.), Lecture Notes in Computer Science, vol 432, pp 88-105, Springer-Verlag (1990)

[Consel & Danvy 91] Charles Consel & Olivier Danvy: *Static and Dynamic Semantics Processing*, in proceedings the *ACM Symposium on Principles of Programming Languages*, January 1991 (1991)

[Consel & Khoo 91] Charles Consel & Siau Cheng Khoo: *Parameterized Partial Evaluation*, in *SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991, Toronto, Canada. Sigplan Notices, vol 26, no 6, pp 92-105 (June 1991)

[Cousot & Cousot 77] Pierre Cousot & Radia Cousot: *Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in *Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, January 1977*, pp 238-252. ACM (1977)

[Danvy & Malmkjær 91] Olivier Danvy & Karoline Malmkjær: *Preprocessing by Specialization*, draft paper, Kansas State University (1991)

[Ershov & Itkin 77] A. P. Ershov & V.E. Itkin: *Correctness of Mixed Computation in Algol-like Programs*, Lecture Notes in Computer Science, vol 53 (1977)

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

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

[Futamura & Nogi 88] Yoshihiko Futamura & Kenroku Nogi: *Generalized Partial Computation*, in [Bjørner, Ershov & Jones 88] (1988)

[Gomard & Jones 89] Carsten K. Gomard & Neil D. Jones: *Compiler Generation by Partial Evaluation*, Information Processing '89, Proceedings of the *IFIP 11th World Computer Congress*, pp 1139-1144, North-Holland (1989)

[Gomard 89] Carsten K. Gomard: *Higher-Order Partial Evaluation – HOPE for the λ-calculus* Master's thesis, DIKU, University of Copenhagen, Copenhagen, Denmark (October 1989)

[Gomard 90] Carsten K. Gomard: *Partial Type Inference for Untyped Functional Programs*, in proceedings of *1990 ACM Conference on Lisp and Functional Programming*, Nice, France, pp 282-287 (June 1990)

[Gomard & Sestoft 91] Carsten K. Gomard & Peter Sestoft: *Globalization and live variables*, in *Preliminary Proceedings, Fifth Glasgow Functional Programming Workshop, Isle of Skye, Scotland, August 1991*, pp 125-149. Department of Computing Science, University of Glasgow, Scotland (1991)

[Haraldsson 77] A. Haraldsson: *A Program Manipulation System Based on Partial Evaluation*, Dissertation, Linköping University, Sweden (1977)

[Henglein 91] Fritz Henglein: *Efficient Type Inference for Higher-Order Binding-Time Analysis*, in proceedings of *Functional Programming Languages and Computer Architecture*, Cambridge, Massachusetts, August 1991, J. Hughes (ed.), Lecture Notes in Computer Science, vol 523, pp 448-472, Springer-Verlag (1991)

[Holst & Gomard 91] Carsten K. Gomard & N. C. K. Holst: *Partial Evaluation is Fuller Laziness*, in proceedings of *Partial Evaluation and Semantics-Based Program Manipulation*, New Haven, Connecticut, June 1991. Sigplan Notices, vol 26, no 9 (September 1991)

[Jensen 90] T. P. Jensen: *Context Analysis of Functional Programs*, speciale, DIKU, Department of Computer Science, University of Copenhagen, Denmark (January 1990)

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

[Jones 88] Neil D. Jones: *Automatic Program Specialization: A Re-examination from Basic Principles*, from [Bjørner, Ershov & Jones 88] (1988)

[Jones *et al.* 89] Neil D. Jones, Peter Sestoft & Harald Søndergaard: *Mix: A Self-Applicable Partial Evaluator for Experiments in Compiler Generation*, Lisp and Symbolic Computation, vol 2, no 1, pp 9-50 (1989)

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

[Kleene 52] S. C. Kleene: *Introduction to Metamathematics*, D. van Nostrand, Princeton, New Jersey (1952)

[Mogensen 88] Torben Æ. Mogensen: *Partially Static Structures in a Self-Applicable Partial Evaluator*, in [Bjørner, Ershov & Jones 88], pp 325-347 (1988)

[Mogensen 89a] Torben Æ. Mogensen: *Binding Time Aspects of Partial Evaluation*, PhD thesis, DIKU, Department of Computer Science, University of Copenhagen (March 1989)

[Mogensen 89b] Torben Æ. Mogensen: *Binding Time Analysis for Polymorphically Typed Higher Order Languages*, in *International Conference on Theory and Practice of Software Development*, Barcelona, Spain, J. Diaz and F. Orejas (eds.), Lecture Notes in Computer Science, vol 352, pp 298-312, Springer Verlag (March 1989)

[Nielson 82] Flemming Nielson: *A denotational framework for data flow analysis*, Acta Informatica, vol 18, pp 265-287 (1982)

[Nielson & Nielson 86] Flemming Nielson & Hanne Riis Nielson: *Semantics Directed Compiling for Functional Languages*, in proceedings of the *1986 ACM Conference on Lisp and Functional Programming*, Cambridge, Massachusetts (August 1986)

[Rees & Clinger 86] Jonathan Rees & William Clinger (eds.): *Revised^3 Report on the Algorithmic Language Scheme*, Sigplan Notices, vol 21, no 12 pp 37-79 (December 1986)

[Sestoft 86] Peter Sestoft: *The Structure of a Self-Applicable Partial Evaluator*, in *Programs as Data Objects*, Copenhagen, Denmark, 1985, H. Ganzinger and N.D. Jones (eds), Lecture Notes in Computer Science, vol 217, pp 236-256 (1986)

[Shivers 88] Olin Shivers: *Control flow analysis in Scheme*, in *Sigplan Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. Sigplan Notices, vol 23, no 7, pp 164-174 (June 1988)

[Talcott 85] Carolyn Talcott: *The essence of $\mathcal{R}$um: A Theory of the Intensional and Extensional Aspects of Lisp-type Computation*, Ph. D. thesis, Dept. of Computer Science, Stanford University, Stanford, California (August 1985)

[Turchin 86a] Valentin F. Turchin: *Program Transformation by Supercompilation*, in *Programs as Data Objects*, Copenhagen, Denmark 1985, H. Ganzinger and N.D. Jones (eds), Lecture Notes in Computer Science, vol 217, pp 257-281, Springer-Verlag (1986)

[Turchin 86b] Valentin F. Turchin: *The Concept of a Supercompiler*, in *ACM Transactions on Programming Languages and Systems*, vol 8, no 3, pp 292-325 (July 1986)

[Turchin 88] Valentin F. Turchin: *The Algorithm of Generalization in the Supercompiler*, in [Bjørner, Ershov & Jones 88], pp 531-549 (1988)

[Wadler 88] Phil Wadler: *Deforestation: Transforming programs to eliminate trees*, in *2nd European Symposium on Programming*, Harald Ganzinger (ed.), Lecture Notes in Computer Science, vol 300, Springer-Verlag (1988)

[Weiner 73] Peter Weiner: *Linear Pattern Matching Algorithms*, IEEE Symposium on Switching and Automata Theory, vol 14 pp 1-11, IEEE, New York (1973)