Compiler Generation by Partial Evaluation

Speciale

Jesper Jørgensen
DIKU, Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100, Copenhagen Ø, Denmark
Electronic mail: knud@diku.dk

January 14, 1992

Abstract

In this report describes techniques required to generate efficient compilers for realistic languages by partial evaluation and to what extent these techniques can be automated. It also describes a large application where a realistic compiler was generated for a strongly typed lazy functional language.

Compiler generation is often emphasized as being the most important application of partial evaluation, but most of the larger practical applications have been outside this field. Especially, no one has generated compilers for languages other than small example languages.

It is well known how compilers can be generated from interpreters by partial evaluation. So we show how to obtain interpreters (in strict functional languages) from formal language descriptions. One way is to specialize a meta-interpreter with respect to language definitions to generate interpreters. This meta-interpreter approach has the advantage that one may define new and better definition languages without having to write new partial evaluators.

We have studied what kind of binding time improvements of interpreters are needed to obtain compilers which are both efficient and generate good target code.

Furthermore, we look at how to modify interpreters to generate optimizing compilers (in the usual sense used in compiler technology).

The language used in the application is called BAWL, and was modeled after the language in Bird and Wadler [Bird and Wadler 1988]. It is a combinator language with pattern matching, guarded alternatives, local definitions and list comprehensions. We described how the general techniques are used to make the compiler generate small and efficient target programs. Various aspects of the performance of the compiler are studied, and the compiler is compared with two compilers for similar languages.

The compiler is reasonably efficient and generates target programs of a quality that can compete both in speed and storage usage with programs produced by a commercial compiler.

Preface

> There is a theory which states that if ever anyone discovers exactly what the Universe is for and why it is here, it will instantly disappear and be replaced by something even more bizarre and inexplicable.
> There is another theory which states that this has already happened. *D. Adams*[1]

This report is a "Speciale" (Master's Thesis) in "Datalogi" (Computing Science) at DIKU (Department of Computer Science, University of Copenhagen) 1991.

The report is submitted as an entry to the University of Copenhagen Prize Contest 1991. The title of the exercise is "Optimization in compilation of higher order programming languages". The text of the exercise (in danish) is included in Appendix G.

This work evolved from a joint project with Lars Mathiesen (DIKU). The contribution of Lars Mathiesen to the present project is explicitly marked below.

This report is the work of the author alone.

The original goal of the project was to try to generate a realistic compiler for a large language by partial evaluation. The language chosen was a lazy functional language which resembles Orwell and Miranda[2], which was eventually named BAWL[3]. The plan was to write a parser using YACC (this was written by Lars Mathiesen and is not described in this report), write an interpreter for the language in Scheme, and then generate the compiler using the Similix partial evaluator. It was never a part of the plan to write a type checker, since this was not the important issue, and therefore the interpreter was written as if no type checker was available, but still assuming that programs are well-typed.

Having achieved this goal we set out to find general ideas and methods involved in the process of compiler generation by partial evaluation and to formalize and prove correct some of these. This became Part 1 of this project. Part 2 describes the original compiler project.

Similix has been updated several times during this work (the most extensive one was the update from version 3.0 to version 4.0), but all that is presented in this report is up to date with version 4.0. By the time this report is completed Similix has been updated to handle partially static data structures and some of the ideas presented in this report are therefore not be relevant to Similix at that time.

We have tried to keep the presentation reasonably self-contained. Some knowledge of partial evaluation, denotational semantics and lazy functional languages will however be beneficial to fully understand the contents of certain chapters. Especially, some ideas presented in [Jørgensen 1991] have been extended and are not described in detail. It should be possible to read Part 2 without first reading Part 1 if one is ready to accept facts from Part 1 face-value.

[1] From D. Adams: "The Restaurant at the End of the Universe" Pan Books Ltd. 1980

[2] Miranda is a trademark of Research Software Ltd.

[3] The origin of this name has absolutely nothing to do with the author becoming a father during the project; it is simply an acronym for Bird And Wadler Language and is due to Kristoffer H. Rose.

An extended abstract of the second part of the report, including some of the basic ideas from the first part of the report, has been accepted for publication[4] [Jørgensen 1992] and will be presented at POPL '92.

Acknowledgements

Carsten Gomard and Peter Sestoft have been my supervisors on the project and have as such done an excellent job.

Anders Bondorf, apart from being one of the creators of Similix[5] and a good friend, deserves a particular thanks for many productive discussions and suggestions.

Also Neil D. Jones deserves a special thanks for creating such an inspiring and lively milieu at DIKU and for employing me on the Semantique project which gave me a lot of computer science experience and self confidence.

Many other people have contributed in various ways; but I would especially like to thank John Hannan, Fritz Henglein, John Launchbury, Lars Mathiesen, Torben Mogensen, Kristoffer H. Rose and Mads Tofte.

I would also like to thank the members of the "TOPPS" group at DIKU.

Finally I would like to dedicate this work to my daughter Mette and to her mother Anne-Marie.

Jesper Jørgensen
Copenhagen, November 1991

[4]The work of writing this abstract was supported by ESPRIT Basic Research Actions project 3124 "Semantique", but not the work presented in this report.

[5]The other one is Olivier Danvy.

Contents

Chapter 1

Introduction

Compiler generation is often emphasized as being the most important application of partial evaluation, but most of the larger practical applications have been outside this field. Especially, no one has generated compilers for languages other than small example languages. In this project we have studied compiler generation by partial evaluation in details. We have done so in order to discover what techniques are required to generate efficient compilers for realistic languages by partial evaluation and to what extent these techniques can be automated.

1.1 Background

A compiler generator is a program that given some machine readable formal definition of a programming language produces a compiler for that language. The research in automatic compiler generation dates back to the 1960s. The first successful compiler generators were little more than parser generators. It is true that YACC[Johnson 1975] can be used to generate complete compilers, but larger portion of the compiler still had to be written by hand and included in the definition, e.g. code for generation. These compiler generators may be called "syntax-directed".

Proper compiler generators should generate complete compilers from formal language definitions. Compiler generator systems of this kind are then called "semantics-directed". The early "semantics-directed" compiler generator systems used some kind of denotational semantics as formal definition language. The basic idea is that a denotational semantics defines a translation from source language syntax into a semantic language (applied lambda calculus) which we can give an operational semantics. In this way we can compile a program by translating it into a lambda calculus expression. This expression may then be simplified according to certain rules (beta-reduction, constant folding, etc.). The final simplified expression is then the "target program" produced by the compilation process. One of the first systems based on this method is Mosses' system SIS [Mosses 1975]. A similar approach was taken by Paulson in his system PCG [Paulson 1982]. Paulson describes his simplifications and then concludes: "Taken together, these improvements cause simplification to resemble symbolic execution of the expressions, ...". But symbolic execution is what partial evaluation is about.

Partial evaluation is a program transformation principle and a program that does partial evaluation is called a partial evaluator. A partial evaluator takes a program and parts of it input and it produces a new specialized program, called a residual program. This residual program will produce the same result when run on the remaining part of the input as the original program would on the entire input, but hopefully more efficiently. The language handled by a partial evaluator is called the subject language of the partial evaluator. A partial evaluator that is written in its own subject language is called self-applicable. One way to use partial evaluation is to write a general purpose program and then specialize this to get programs for specific cases.

A problem with the early compiler generator systems was that they interpreted the language

definitions (or some modified form of the language definitions), that is, no specialized compilers were generated. This is where partial evaluation improves on these earlier methods. A partial evaluator can be viewed as a *general purpose compiler*. It can take a language definition in the form of an interpreter and a program in the language being defined as input. It can then specialize the interpreter with respect to the program and produce a specialized version of the interpreter. This specialized version of the interpreter computes the same function as the original program, but is written in subject language of the partial evaluator. In other words, it is a target program. So a partial evaluator can compile much in the same way early compiler generator systems can, but partial evaluators may do more. If we have a self-applicable partial evaluator we can generate compilers. In other words, it can remove the overhead introduced by the interpretation of the language definitions and produce efficient compilers. We simply specialize the partial evaluator (the general compiler) with respect to the interpreters to get compilers.

In this way we regard the subject language of the partial evaluator as our formal definition language. Since partial evaluators, at least today, exist only for ordinary programming languages, this means that we are restricted to using these languages for our formal definitions if we want to use partial evaluation to generate compilers. It would be nice if we could use a more readable formal definition language without having to write a partial evaluator for the language. This is in fact possible. We write a *general purpose interpreter*, called a *meta-interpreter*, that can interpret any language given a formal definition of the language. If we write the meta-interpreter in a language for which we have a self-applicable partial evaluator, we can first specialize the meta-interpreter with respect to a definition for a given language. This will give us an interpreter for the language, and from this we can generate a compiler in the same way as described above.

One of the original goals of automatic compiler generation was to make it easier to design, document and implement programming languages. That is, language designers should be able to use a compiler generation system to produce prototype implementations of the languages they are working with. It was not one of the primary goals to make production quality compilers. With the first "semantics-directed" compiler generation systems, the quality of the produced compilers were also very poor compared with handwritten ones. What we hope to show in this report is that semantics-directed compiler generation is not as inefficient as it is sometimes claimed, and that partial evaluation may be one of the key principle to use, if efficient compiler generators are to be constructed.

1.2 About the project

In this project we have mostly been looking at the direct way of doing compiler generation by partial evaluation: by writing language specifications as interpreters directly in the subject language of the partial evaluator. But our final goal was to find a method to generate these interpreters automatically from language definitions written in a better meta-language than conventional programming languages. We believe to have succeeded in achieving this goal.

Our first approach was to define a translation from denotational semantics into a strict functional language, such as Scheme. This led us to constructing a meta-interpreter for a small denotational definition language and we were able to generate compilers for small languages using the approach discussed above.

We have also generated a compiler for a realistic language by hand-translating a denotational semantics for the language to an interpreter, then using partial evaluation to get the compiler. The language used is a lazy combinator language resembling Miranda[1].

A common experience among people working with partial evaluation is that not all programs

[1]Miranda is a trademark of Research Software Ltd.

are equally well suited for partial evaluation and that a certain amount of rewriting is necessary to make a program "partially evaluate well." This rewriting is usually called *binding time improvements*. We have therefore looked at what kind of binding time improvements we need to get interpreters that "partially evaluate well" and thus compilers that generate good target code.

We have also looked at how to make the generated compilers do optimizations (in the usual sense used in compiler technology: peephole optimizations, strictness optimizations, etc.). It turns out that it is possible to do many of these optimizations by having the interpreter collect information during evaluation and then use the information to do "optimizations". This does not in general result in a faster interpreter, since it will often take longer to collect the information than the time saved by the optimization. But by generating a compiler from the "optimized" interpreter we may get an optimizing compiler, if the collection of information and the optimizations based on the information can be performed at compile time (during specialization). Not all optimizations can be performed in this way and we have therefore looked at alternative ways to do these.

The final "improved" and "optimized" compiler is reasonably efficient and generates target programs of a quality that can compete both in speed and storage usage with programs produced by commercial quality compilers.

A prototype system to compile, run and test programs has been written. It works as a top level interpreter loop that prompts for a command to execute or an expression to evaluate. This system has been used to test, debug and demonstrate the compiler with great success. A description of this system is not included in this report.

1.3 Overview of the report

The first part of this report describes the techniques required to generate compilers by partial evaluation. In Chapter 3 we introduce the basic ideas, and show how partial evaluation comes into play. We then turn to the problem of obtaining an interpreter from denotational semantics. In Chapter 4 we give a short introduction to the style of denotational semantics that we intend to use. In Chapter 5 we show how to translate a denotational semantics into an interpreter written in a strict functional language and how to ensure that the resulting interpreter is correct with respect to the semantics. For this we describe a translation from a lambda calculus based language with both strict and non-strict abstractions into a strict sublanguage (i.e. one without non-strict abstractions). This translation will be based on a type inference system. We discuss how to use this translation to translate a denotational semantics into an interpreter written in Scheme. This also includes a proof of an applicative fixed point operator that has to be used for local recursion. In Chapter 6 we discuss a method for automating the translation of denotational semantics to interpreters using meta-interpreters. In Chapter 7 we discuss what kind of binding time improvements are needed in order to get good compilers (generating small and efficient target programs). In Chapter 8 we discuss the kind of optimizations that can be incorporated into the compilation process by changing the interpreters. In Chapter 9, the last chapter of the first part of the report, we discuss the various ideas presented and possible future work.

The second part of the report describes a large application of the techniques described in the first part. This is the generation of a realistic compiler for a strongly typed lazy functional language. The language, which was called BAWL, is modeled after the language used by Bird and Wadler [Bird and Wadler 1988]. It is a lazy combinator language with pattern matching, guarded alternatives and local definitions. In Chapter 10 we introduce BAWL and describe the syntax of the language, and in Chapter 11 we present a denotational semantics for BAWL. In Chapter 12 we show how an interpreter can be obtained from this semantics. We also show how laziness is introduced and how to add a print-routine. This first interpreter was

specialized to get a first version of a compiler for BAWL and in Chapter 13 we give examples of binding time improvements that we performed on this interpreter. In Chapter 14 we show what kind of "optimizations" we have incorporated into the final version of interpreter to obtain an optimizing final version of the compiler. From this improved version of the interpreter a compiler for BAWL was obtained and in Chapter 15 we give an example of a target program produced by this compiler. In Chapter 16 we compare the performance of the two compilers and we compare the final compiler with two widely used compilers for similar languages: Miranda and LML. In Chapter 17 we discuss what further improvements could be made to the final version of the compiler and we conclude.

Chapter 2

Preliminaries

Purpose of this chapter is to give an introduction to the basic ideas used in this thesis and introduce the main tool Similix. The most important idea is partial evaluation and we have therefore dedicated the next section solely to this subject. The partial evaluator we have used is Similix, an autoprojector (self applicable partial evaluator) for a subset of Scheme. Similix also deserves a whole section and will be described in Section 2.2. Denotational semantics also play a significant role and will be discussed in more details in Chapter 4, but for those only interested in reading the second part of the report and who are familiar with denotational semantics, we round this chapter off with a short introduction to the style we use. This may be found in Section 2.3. The last important ingredient is functional programming languages, both strict and lazy. We will not discuss these here (excellent introductions can be found elsewhere, e.g. [Dybvig 1987] and [Bird and Wadler 1988]), but hope that the reader has courage and will to see her/his way through the abominable syntax of Scheme with its bulks of parentheses; Scheme also has its good sides.

2.1 Partial Evaluation

Partial evaluation, also called program specialization, is a program transformation principle. A program that does partial evaluation is called a *partial evaluator* or a *specializer*. It takes a program and part of the programs input, and produces from this a new program, called a *residual program*. This residual program will, when run on the remaining input, produce the same result as the original program run on the entire input. The point is that the partial evaluator will try to do as much work as possible with the part of the input that is available at *partial evaluation time* and produce a residual program that is as efficient as possible. Hence, a partial evaluator is an implementation of Kleene's S_n^m theorem, but a non-trivial one. Depending on the language being partial evaluated, one may view partial evaluation as either a special form of evaluation in which values can be both usual values or program text, or as a special reduction strategy.

There are several ways in which partial evaluation may be used. If a program is going to be run several times with only part of its input changing, then time may be saved by first partially evaluating the program with respect to the constant part of the input to get a hopefully much faster residual program. This residual program can then be run on the remaining changing part of the input. But even if a program is only going to be run once, partial evaluation may speed up execution. This is because during standard evaluation the same function may be called many times with the same values of some of the arguments and the computation depending on these arguments are then performed each time. Partial evaluation may sometimes ensure that this computation is only performed once. Good examples of this are Consel's and Danvy's KMP example [Consel and Danvy 1989] and Mogensen's ray tracing example [Mogensen 1986]. This also corresponds to the normal experience, that it is often faster first to compile a program and

then run the program than it is to interpret the program.

For historical reasons partial evaluators are usually called Mix. The above informal description of partial evaluators can be re-stated more formally by the *mix equation*:

$$\mathrm{L}_1\ P(d_1, d_2) = \mathrm{L}_1\ (\mathrm{L}_2\ Mix(P, d_1))\ d_2 \tag{2.1}$$

where L_1 is the programming language[1] that Mix handles, L_2 the language that Mix is written in, P is a L_1-program, d_1 the known part of the input and d_2 the unknown part. The mix equation is in fact just the correctness criterion for a partial evaluator. The known input is normally referred to as the *static* input and the unknown input as the *dynamic* input. If $\mathrm{L}_1 = \mathrm{L}_2$ the partial evaluator is said to be *self-applicable* which is important for compiler generation as we shall see in Chapter 3. A self-applicable partial evaluators is also called an *autoprojector*.

The first self-applicable partial evaluator was Mix [Jones *et al.* 1985]. Today self-applicable partial evaluators are available for many types of languages: imperative languages (a flowchart language [Gomard and Jones 1989]), functional languages (Mixwell [Jones *et al.* 1985], Scheme [Consel 1988], [Bondorf 1990a]), logic programming languages (Prolog [Bondorf and Mogensen 1990]).

Partial evaluators often use *binding time analysis* to determine prior to specialization which parts of a program are static and which are dynamic. That a part of a program is static means that it can be completely evaluated during specialization. If this is not the case then the part is said to be dynamic. This means that there are usually two *binding time values*: static (S) and dynamic (D). If the partial evaluator handles higher order functions, side effects or *partially static data structures* (structures that can have both static and dynamic substructures) there is a need for other binding time values.

Binding time analysis is considered important for achieving efficient self-application and therefore also for compiler generation [Bondorf *et al.* 1990].

2.2 Similix

The partial evaluator used in this project is Similix, though many of the methods presented may apply to other partial evaluators as well. This section describes some aspects of Similix (version 4.0) that are important for understanding the ideas presented in this report. Similix was created by A. Bondorf and O. Danvy and extended to include higher order function by Bondorf. A description of Similix can be found in [Bondorf and Danvy 1991] [Bondorf 1991a]. In short, Similix is an autoprojector for large subset of Scheme [Rees and Clinger 1986], including both abstract data types, lambda abstractions and side effects on global variables.

When specializing a program with Similix one has to specify which function, called the *goal function*, in the program is going to be specialized and some static input to this function.

Similix uses *monovariant* binding time analysis which has the effect that residual functions are only generated for one set of binding time values for the arguments.

Similix (version 4.0) has no *partially static data structures*. This means that Similix (version 4.0) does not use reduction rules like:

$$\texttt{(car (cons } expr_1\ expr_2\texttt{))} \Rightarrow\ expr_1$$

The reason for this is that it is hard to handle such transformations in a semantically safe way in a strict language. However, a way to do this has been found cf. [Bondorf 1992]. If evaluation of

[1] A programming language is a partial function L: $L \rightarrow D^* \rightarrow D$ where L is the set of L-programs. If P is an L-program then the intention is that L P: $D^* \rightarrow D$ is the input-output function denoted by P. The notation L P d then means the result of running the L-program P on data d.

$expr_2$ may not terminate then this implies that the left hand side of the rule may not terminate, but the same does not hold for the right hand side.

Since Similix handles higher order functions and side effects the binding time domain of Similix contains two additional values: closure (CL) meaning that a part of a program with this binding time value is a higher order function that can be applied at *specialization time* (during partial evaluation), and external (X) meaning that a part of a program with this binding time value may perform side effects. We will sometime when speaking about *binding time improvements* (see Chapter 7) loosely talk of S and CL as static values since these will be processed at specialization time, and D and X as dynamic since these will not be processed at specialization time.

Similix also allows users to introduce primitive operations, called abstract data operators and defined in separate files called adt-files. The actual representation of the primitive operations is hidden from the specializer. This means that all the specializer can do with a call to one of these operations is to completely evaluate the call (if all arguments are static) or generate a residual call to the operation. This has the advantage that these primitives can be written in full Scheme and that one can introduce side effects this way. Actually all the usual standard functions in Scheme (`cons, car, cdr, equal?, display`, etc.) are defined as such primitive operations.

2.3 Denotational Semantics

In semantic definitions, our style is close to that of Schmidt [Schmidt 1986]. The conditional is written $\_ \rightarrow \_ \,[]\, \_$, and function update $[\mathrm{i} \mapsto \mathrm{v}\,]\rho$ is short for $\lambda \mathrm{i}'.\mathrm{i}{=}\mathrm{i}' \rightarrow \mathrm{v} \,[]\, \rho\ \mathrm{i}'$. Our notation differs slightly from that of Schmidt in that we allow *semantic parameters* (ranging over semantic domains) to valuation functions to be written on the left hand side, while Schmidt always write these using lambda's on the right hand side. We regard this purely as a syntactical difference since it will always be clear which are syntax parameters and which are not. We also use certain abbreviations: if v is a variable ranging over a syntax domain V, then $v*$ is a variable ranging over V*, where V* is the domain of sequences from V. A short introduction to the style of denotational semantics we use can be found in Chapter 4.

We some times leave out tagging and tag checking of values in sum domains when giving meaning well typed programs in statically typed languages, but only when the tags may be inferred and reinserted (e.g. not for list types). We make this more clear in Chapter 4.

Readers familiar with denotational semantics should have no problem reading the denotational semantics of BAWL found in Chapter 11.

Part I

Compiler generation by partial evaluation

Chapter 3

Introduction to Compiler Generation

In this chapter we give some background on compiler generation and an overview of our approach to this subject. We also describe the idea behind compiler generation by partial evaluation.

3.1 Background

A compiler generator is a program (or system) that given some machine readable formal description of a programming language produces a compiler for that language. The formal description can be of many forms: a denotational semantics, an attribute grammar, a combinator based semantics, some form of an operational semantics, etc.

Automatic compiler generation is an old research topics dating back to the 1960s. The first attempts were not really true compiler generators, but rather parser generators and this kind of compiler generation is usually called "syntax-directed". The code generation still had to be written by hand. The experiments with "syntax-directed" compiler generations have been successful and have resulted in efficient tools to generate parsers etc. LEX [Lesk and Schmidt 1975] and YACC[Johnson 1975].

But a proper compiler generator should generate an entire compiler including parser, code generator, etc. The kind of compiler generation that include code generation is usually called "semantics-directed". Semantics-directed compiler generation may cover more than just code generation, but it will mainly be this part that we will be concerned with. A complete compiler generator system could be obtained by combining our methods with those of "syntax-directed" compiler generation.

One of the first attempts to base a compiler generator on a denotational semantics was Peter Mosses' SIS [Mosses 1975][Mosses 1979]. SIS was slow (2 minutes to calculate the factorial of 3), but was on the other hand primarily intended to make it possible to run programs according to their specifications.

Paulson's PCG [Paulson 1982] uses an attribute grammar and his compilers could for example, do compile-time type checking.

CERES, by Christiansen and Jones [Christiansen and Jones 1982], uses a combinator based semantics, where the set of combinators is not fixed, but can be defined independently for each application. CERES has been used to generate compilers on small test languages.

The Quokka System [Vickers 1986] takes a denotational semantics and a syntax definition as input and produces a complete compiler that can parse source programs and translate these into lambda expressions. These lambda expressions can then either be interpreted or compiled further into PASCAL. Specifications can have access to user-written PASCAL functions and

procedures.

Weis's SAM system [Weis 1987] also uses a denotational semantics description. With SAM, Weis has generated both small and very efficient target programs.

3.2 Compiler Generation using Partial Evaluation

Partial evaluators are not compiler generators, but one of the primary examples of their uses has certainly been compiler generation. This is also supported by the fact that one of the main motivations for making partial evaluators self-applicable is that this makes compiler generation possible. The way we propose to generate compilers in this thesis deals with more than just partial evaluation, but partial evaluation is just a central tool to which most of the other ideas have to adjust.

Roughly speaking compilers consist of a parser and a code generator (we will not consider static semantics processing, that is, type checking etc. in this thesis). So to be able to generate a compiler for some language automatically we have to specify a formal description of both the syntax and the semantics of the language. The parser is generated from the description of the syntax, and techniques for this are well known. The code generator on the other hand is what we can generate by partial evaluation. Figure 1 shows the whole process of compiler generation.

In this figure texts in boxes represent operations (either manual or automatic) and texts in boxes with rounded corners data objects handled by the operations. The parser and the code generator are both data objects and operations and are therefore placed in boxes of both kinds.

The chapters referred to describe the corresponding operations in details. The left part of the diagram shows the parser generation and is not explained further in this report. It is only shown for the sake of completeness. The operation called "Translation" is a translation from a denotational semantics into an interpreter written in Scheme. The important part of this is a translation from call by name lambda calculus to call by value lambda calculus, but the operation may include more than this which we will show in Chapter 5. The binding time improvements translate the interpreter into a new interpreter that ensures that the final compiler will do more compile time operation, thus producing better target code. The optimizations are other improvements of the generated compiler obtained by using simple analysis in the interpreter that, for instance collects and use certain information during interpretation. This will be explained in details in Chapter 8. Cogen is the compiler generator of the partial evaluator which will be described in the next section.

All except the last step leading to the code generator is presently done by hand, as exemplified in the second part of this report, but we shall discuss possible way to automate these.

3.3 Compiler Generation by Partial Evaluation

In this section we will explain the fundamental ideas behind compiler generation using partial evaluation. Namely how partial evaluation comes into play.

3.3.1 The Usual Method

We will now explain how one may compile and how one may generate compilers by partial evaluation. Assume that we have written an interpreter Int for some language M. Let P be an M-program and d some input to P. By using the Mix equation (1.1) we get:

$$\mathrm{L}\ Int(P,d) = \mathrm{L}\ (\mathrm{L}\ Mix(Int,P))\ d$$

Let now $Target$ be defined by:

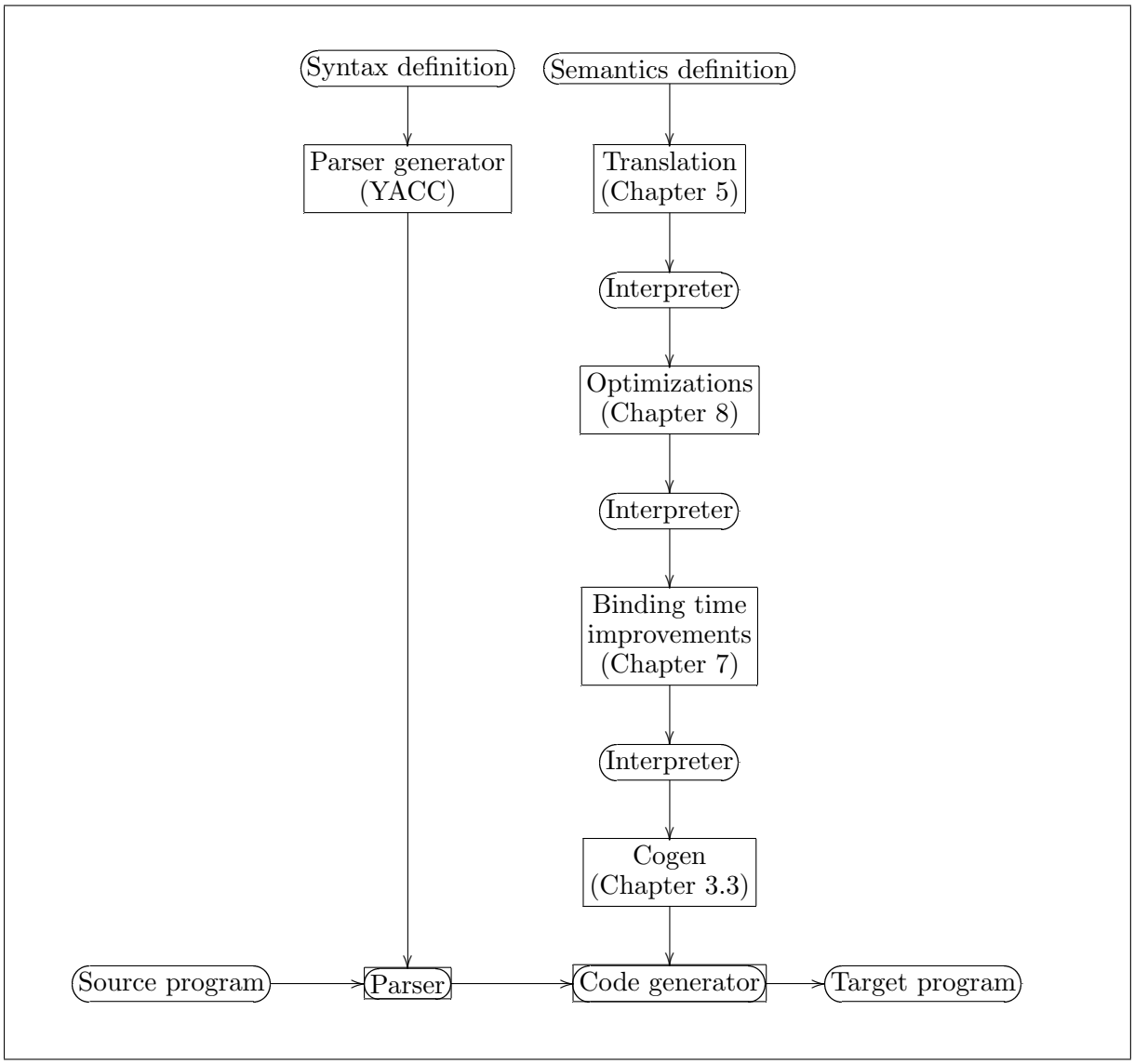

Figure 1: Overview of the compiler generation process

$$Target = \text{L } Mix(Int, P) \tag{3.1}$$

We can then see that $Target$ is a L-program computing the same function as P and we therefore say that we have compiled P into $Target$. Using the Mix equation once again we get:

$$Target = \text{L } Mix(Int, P) = \text{L } (\text{L } Mix(Mix, Int))\ P$$

From which we can see that (L $Mix(Mix, Int)$) must be a compiler from M to L, since it produces $Target$ when run on P. That is we have:

$$Compiler = \text{L } Mix(Mix, Int) \tag{3.2}$$

We can now use the Mix equation a third time:

$$Compiler = \text{L } Mix(Mix, Int) = \text{L } (\text{L } Mix(Mix, Mix))\ Int$$

From which we can see that (L $Mix(Mix, Mix)$) is a compiler generator, since it produces $Compiler$ when run on Int. We call this compiler generator $Cogen$, and we have:

$$Cogen = \text{L } Mix(Mix, Mix) \tag{3.3}$$

The equations 2.1 to 2.3 is usually called the Futamura projections.

This means, in terms of compiler generation, that the specification is the interpreter and the specification language is the source language of the partial evaluator. It also means that the choice of partial evaluator will fix the specification language.

3.3.2 An Alternative Method

The way to do compiler generation by partial evaluation described above is maybe the most direct, but not the only natural way. An other way is using a meta interpreter. A meta interpreter $Meta$ is a program that takes a language definition Def and a program P, and then runs the program P as a program in the language defined by Def. We can write this as:

$$output = \text{L } Meta(Def, (P, input))$$

This means that we can get an interpreter for the language defined by Def by specializing $Meta$ with respect to Def:

$$Int = \text{L } Mix(Meta, Def)$$

or by using $Cogen$ we can get an interpreter generator:

$$Intgen = \text{L } Cogen\ Meta$$

Now that we have an interpreter it is easy to get a compiler for the language Def:

$$Compiler = \text{L } Cogen(\text{L } Intgen\ Def)$$

This method has some advantages compared to the one previously described. Firstly, we are free to define our own definition language, especially one that is more readable than e.g. a Scheme interpreter. Secondly, we can write Meta in such a way that the user of the compiler generation system is free from the problem of having to do binding time improvements on the language definition (the interpreter).

In this thesis we will mostly be concerned with methods related to the direct way of doing compiler generation by partial evaluation describe in Section 3.3.1, but we will return to the method using meta-interpreters in Chapter 6.

Chapter 4

Denotational Semantics

In the next chapter we will be demonstrating how to translate a denotational semantics into an interpreter. For this purpose we will give a short overview of the style of denotational definitions we intend to use. We will formalize as much as will be needed in the next chapter and let the rest of the description be informal. The interested reader may refer to e.g. Schmidt for a more detailed introduction.

Denotational semantics is a formalism for assigning meaning to programming languages. The meaning of a programming language can be expressed as a (partial) function which maps programs into mathematical functions and this also what denotational semantics does. Denotational semantics handles undefinedness by introducing a new element $\perp$ and thus making semantic functions total.

We will use denotational semantics in a style close to that of Schmidt [Schmidt 1986]. Following Schmidt a denotational description of a language consist of three things: an abstract syntax, a collection of semantic algebras and a collection of valuation functions. An example of a denotational semantics may be found in Figure 14 of Chapter 5.

4.1 Abstract Syntax

The definition of the abstract syntax of a programming language is given in *Backus-Naur form* (BNF) notation. An abstract syntax describes structure. The nonterminals are, contrary to a concrete syntax, no longer symbols, but *words* (also sometimes called *tokens*). This is natural, since it will be an entire structure that the semantics will assign meaning to, not the individual symbols. We sometimes use a more verbose form of abstract syntax, where we use more than one word to express an syntactical form, but it will always be implicit that we might as well have used one. For example, we may write the abstract syntax of a let-expressions in some functional language as:

`let` *var* `=` $expr_1$ `in` $expr_2$

instead of the more formal, but less informative:

`let` *var* $expr_1$ $expr_2$

We will not go into details with the notation of abstract syntax; descriptions of this can be found in many other texts.

4.2 Semantic Algebra

A semantic algebra is a formalism for introducing domains and operations (functions) on these domains. The domains can be built up from primitive domains (sets)[1] such as the natural numbers, the integers, etc., and domain constructors, which build up compound domains. The standard domain constructors are: the product domain constructor $\times$, the sum domain constructor $+$, the function domain constructor $\rightarrow$ and the lifted domain constructor $\_\perp$. All these domain constructors have operations associated with them which can be used to build up operations over the constructed domains. We will not describe these operations in details, since they are fairly standard and may be found in any book on denotational semantics [Schmidt 1986]. New domains can also be defined by recursive domain equations, but we will not here discuss how these are to be interpreted. The interested reader is referred to Schmidt Chapter 11. An example of a semantic algebra can be seen in Figure 2.

Semantic algebra:

Domains:
$\rho \in \text{Env} = \text{Id} \rightarrow \text{Value}$ (environments)

Operations:
access: Id $\rightarrow$ Env $\rightarrow$ Value
access = λi.$\lambda\rho.\rho$ i

update: Id $\rightarrow$ Value $\rightarrow$ Env $\rightarrow$ Value
update = λi.λv.$\lambda\rho.[\text{i} \mapsto \text{v}]\rho$

Figure 2: Example of semantic algebra: Environments

4.3 Valuation Functions

This section describes valuation functions. Valuation functions are the function that assign meanings to syntax phrases and entire programs. In a denotational semantics a valuation function is defined for each syntax domain. The definition of a valuation function is a set of equations, defining the value of the function for each syntax phrase of the syntax domain. The definition must be *compositional*, that is, the semantics of a syntax phrase is determined in terms of the semantics of it subphrases. Figure 3 shows the syntax of a definition of a valuation functions.

The function names *fun* are the valuation functions being defined and these can only be applied to abstract syntax. Base values are values of the primitive domains, e.g. $1 \in$ Nat. Variables are semantic variables ranging over semantic domains. In the following A and B are arbitrary domains. The sum domain builder **in***dom* construct elements in sum domains, e.g. **in**A: A $\rightarrow$ A + B. The cases-of-expression takes these apart again. The pairing operation (_,_) builds elements in A $\times$ B and the operations **fst** and **snd** are the corresponding projections on A and B. Operations *op* are defined in the semantic algebra. The double brackets $[\![\ ]\!]$ are used to separate the syntax from the semantics and *asp* is an abstract syntax pattern ranging over abstract syntax trees as defined by the abstract syntax. The fixed point operator **fix** is defined the usual way:

$$\textbf{fix}\ \text{F} = \bigsqcup_{n=0}^{\infty} \text{F}^n(\perp)$$

[1] Primitive domains are just sets, that is, they are not cpo's.

def	::=	*eq* {*eq* }	(definition of valuation function)
eq	::=	*fun*⟦ *asp* ⟧ = *expr*	(equation)
expr	::=	*base*	(base value)
	|	*var*	(variable)
	|	*fun*⟦*asp*⟧	(valuation function application)
	|	λ*var.expr*	(abstraction)
	|	$\underline{\lambda}$*var.expr*	(strict abstraction)
	|	$expr_1$ $expr_2$	(application)
	|	**in***dom*(*expr*)	(sum domain builder)
	|	**cases** *expr* **of** *cl* {▯ *cl* }	(cases-of-expression)
	|	($expr_1$,$expr_2$)	(product domain builder)
	|	**fst** *expr*	(left projection)
	|	**snd** *expr*	(right projection)
	|	*op*	(operation)
	|	**fix** *expr*	(the fixed point operator)
cl	::=	*pat* → *expr*	(clause)
pat	::=	**is***dom*(*var*)	(pattern)
	|	_	(wildcard)

Figure 3: Syntax of valuation functions

There is no conditional, but we will assume that a domain of truth values Bool is defined as Bool = True + False, where True = False = Unit. Then we use the abbreviation: $expr_1$ → $expr_2$ ▯ $expr_3$ for **cases** $expr_1$ **of is**True() → $expr_2$ ▯ **is**False() → $expr_3$. We will also use other abbreviations and these can be seen in Figure 4.

let *var*=$expr_1$ **in** $expr_2$	=	($\underline{\lambda}$*var.*$expr_2$) $expr_1$
$expr_1$ **where** *var*=$expr_2$	=	(λ*var.*$expr_1$) $expr_2$
$expr_1$ → $expr_2$ ▯ $expr_3$	=	**let** t=$expr_1$ **in** **cases** t **of is**True() → $expr_2$ ▯ **is**False() → $expr_3$
expr!*n*	=	**fst** (**snd** (... (**snd** *expr*) ...)) where the numbers of **snd**'s is equal to *n*
($expr_1$,...,$expr_n$)	=	($expr_1$,($expr_2$,(...,$expr_n$)...)) for all n > 1
λ*pat.expr*	=	λ*var.***cases** *var* **of** *pat* → *expr* ▯ _ → ⊥

Figure 4: Syntactic Abbreviations

We will often be a little sloppy using injection tags. We do this because using injection tags often makes the semantic definitions quite unreadable and we can allow us this freedom when we work with typed languages. Here is a small example showing why this is a good idea: Assume that we are writing a semantics for a functional language and we have a semantic domain: Value = (Base + Error + Function)$_\perp$. Then the valuation function may have a case for application that looks like:

$$\begin{aligned}\mathbf{E}[\![(e_1\ e_2)]\!]\rho = \ & \textbf{let } v{=}\mathbf{E}[\![e_1]\!]\rho \textbf{ in} \\ & \textbf{cases } v \textbf{ of} \\ & \quad \text{isFunction(f)} \rightarrow \text{f}(\mathbf{E}[\![e_2]\!]\rho) \\ & \quad [\!] \text{ isBase(b)} \rightarrow \text{inError()} \\ & \quad [\!] \text{ isError()} \rightarrow \text{inError()} \\ & \textbf{end}\end{aligned}$$

This kind of definition soon becomes very cumbersome and since we know that the programs that we will assign meaning to are only the well typed ones, none of the two last cases can ever come into action. We will therefore write the definition as:

$$\mathbf{E}[\![(e_1\ e_2)]\!]\rho = (\mathbf{E}[\![e_1]\!]\rho)\ (\mathbf{E}[\![e_2]\!]\rho)$$

leaving the tagging and untagging implicit. It is in fact possible to infer these operations since the language is typed. One can make this more formal (see [Launchbury 1991] page 39). We also leave out the **let**-expression assuming "application" strict in its first argument.

4.4 How Denotational Semantics Assigns Meanings to Programs

As we have presented denotational semantics until now, it corresponds to that of Scott and Strachey as described for example in [Schmidt 1986]. This means that we regard a denotational semantics as a translation from syntax into a mathematical notation denoting a mathematical object. Thus we can unfold the applications of the valuation functions to their syntax arguments and we can do so because we demand that denotational semantics is compositional. The result of this unfolding will be some expression not containing any program syntax, built up using only the notation of the lambda calculus and the various operations and base values. This expression then represents the meaning of the syntax. We call the language that the denotational semantics is written in for the *meta language* and the object language of the translation the *semantic language*.

We could also impose another interpretation on a denotational semantics, by interpreting the result of the translation differently. If we regard the objects produced by the translation as terms in some form of *applied* lambda calculus. We might specify the meaning of this by giving some form of *operational semantics* for the calculus [Barendregt 1984]. This includes the notion of *reduction strategy* and *normal form*.

If we have a denotational semantics for a given language L and a L-program P, we can execute P by simply translating it into a lambda calculus term and then apply this term to some input and reduce the application to normal form. We might also regard the whole denotational semantics as a program in some language whose semantics is given by the scheme just outlined. In this way a denotational semantics just becomes an interpreter for the language it defines.

We might even write an interpreter for denotational definitions and call this a *meta-interpreter*. This is a program that given a definition of a language *Def*, a program P in the language defined by *Def* and some input to P, translates P according to the definition *Def* and then interprets the resulting term applied to the input. If the program has been parsed (i.e. is syntactically well-formed) we may do the translating by need, that is, interleave the translation and the reduction. This will correspond more directly to interpreting the definition. In Chapter 6 we will describe such a meta-interpreter.

From now on when we speak of a denotational semantics as a program we will call it a *denotational definition* and we will call a language of such denotational definition a *denotational definition language*. The language in Figure 3 will be our example of a denotational definition language.

Chapter 5

Implementation of Denotational Definitions

As we discussed in Section 4.4 of previous the chapter a denotational definition can be viewed as an interpreter. But this does not help us much if we do not have an implementation of our denotational definition language. On solution to this problem would be write such an implementation, and in Chapter 6 we describe how to do so when we discuss meta-interpreters. Another solution would be to translate our definition language into some other language for which an implementation exist. This is what we will describe in this chapter.

We will show how to translate a denotational definition into a strict functional language. The translation of a definition will be an interpreter written in the functional language. We will keep our discussion as general as possible so that as many of the ideas presented will apply to as many different strict functional language as possible. When we have to become specific and give examples we will use Scheme and then only the subset of Scheme that Similix can handle. The reason for using a strict functional language as target language for our translation is of course that we intend to apply Similix to the translated definition.

There are several reasons for wanting to proceed in this way. Firstly, denotational definitions are easier to define language in than a functional programming language. Secondly, we want to be able to do compiler generation by partial evaluation starting with a semantics and ending with a compiler. Thirdly, we want to aim at correctness of implementation. Indeed, if we can show that our translation is correct and we are working with a correct partial evaluator, then we have a way to generate correct compilers (with respect to a given semantics). Finally, if we want to do automatic "semantics-directed" compiler generation it will be important to formalize the translation from our semantic into the language that our partial evaluator handles.

There are several things to consider when translating denotational definitions into a strict functional language. Firstly, we will have to be sure to preserve the call by name semantics of the denotational definition language. Secondly, we will have to decide on some representation of domains in the functional language. Thirdly, if the language does not have pattern matching driven definitions, we will have to translate the pattern matching of the denotational definitions into some equivalent matching code. Methods for this are well known, see for instance Wadler's chapter in [Peyton Jones 1987] or [Jørgensen 1991].

Since our definition language is a functional language the translation should be fairly easy to define, but because of the strictness of of our target language we have to look at simulating call by name in a call by value language. We will start by showing how to do this for a simple subset of our definition language and the in Section 5.4 we will discuss how we can use this to obtain a translation of all of our definition language.

5.1 Simulating Call by Name in a Call by Value Language

In this section we will show how to simulate call by name in a call by value language. We will do this by defining a translation that takes a program in a call by name language (the lambda calculus with call by name semantics) and translates this into an equivalent program in a call by value language (the lambda calculus with call by value semantics). We will prove that the translation is correct with respect to the semantics of the two languages. We will then use the translation as a model for implementing call by name functional languages in strict functional languages.

The idea is to suspend the evaluation of arguments by enclosing them in lambda abstractions. This is a well known idea that at least dates back to the 1960s where Landin used it [Landin 1965]. So, if *expr* is an argument expression in an application, we change this into $(\lambda\alpha.expr)$ which means that the argument is now as much evaluated as the call by value strategy will evaluate it (this is usually referred to as *weak head normal form*). We call $(\lambda\alpha.expr)$ a *suspension*. When, at a later point, the value of the expression *expr* is actually needed we apply the corresponding suspension to some dummy argument `#` and evaluation of the expression starts. We will refer to this as forcing the evaluation.

We will start by introducing the languages that we consider. The two languages have the same syntax, shown in Figure 5.

Abstract syntax:

$e_\tau \in \text{Exp}$, $x_\tau \in \text{Var}$, $\tau, \sigma \in \text{Type}$

$e_\tau ::= x \mid \texttt{\#} \mid \lambda x_\tau.e_\tau \mid e_\tau\ e'_\tau \mid \texttt{rec}\ x_\tau\ e_\tau$

$\tau ::= \text{Unit} \mid \tau \to \tau$

Figure 5: Abstract syntax of the typed lambda calculus

The only base constant in the languages is `#`. The languages have types and all expressions are annotated with their types, that is, we use explicit typing. To emphasize that the lambdas in the strict language are strict we will underline these to get a consistent notational convention. Expressions can be shown to be well typed by the rules in Figure 6. The semantics of the non-strict version of our language is shown in Figure 7.

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x\colon \tau}(\text{Var}) \qquad \Gamma \vdash \texttt{\#}\colon \text{Unit}\ (\text{Const}) \qquad \frac{\Gamma \vdash e_{\tau\to\sigma}\colon \tau \to \sigma \quad \Gamma \vdash e'_\tau\colon \tau}{\Gamma \vdash e_{\tau\to\sigma}\ e'_\tau\colon \sigma}(\to\text{E})$$

$$\frac{[x \mapsto \tau]\Gamma \vdash e_\sigma\colon \sigma}{\Gamma \vdash \lambda x_\tau.e_\sigma\colon \tau \to \sigma}(\to\text{I}) \qquad \frac{[x \mapsto \tau]\Gamma \vdash e_\tau\colon \tau}{\Gamma \vdash \texttt{rec}\ x_\tau\ e_\tau\colon \tau}(\text{Rec})$$

Figure 6: Rules for type checking

Expressions are assumed to be well typed. The constant # is the semantic value of `#`. Figure 7 also defines a semantic operation Lift that we need when proving the correctness of our translation. The semantics of the strict version of our language is almost identical to the semantics of the non-strict one. The only difference is that the equation for abstraction in the call by value semantics is:

$$\mathbf{E}_S[\![\underline{\lambda}x_\tau.e_\sigma]\!] = \underline{\lambda}\rho.\underline{\lambda}\text{v}.\mathbf{E}_S[\![e_\sigma]\!][x \mapsto \text{v}]\rho$$

where $\underline{\lambda}$ is Schmidt's strict lambda. The domains are the same, but we will use different names for the semantic domains to make it clear where values belong. This mean that we use the names

Val_S and Env_S for the value and the environment domains when giving meaning to programs in our call by value language and Val_N and Env_N for the value and the environment domains when giving meaning to programs in our call by name language.

Semantic algebra:

Domains:

$\mathrm{Val}_N = \sum_{\tau \in Type} \mathrm{D}_\tau$
$\mathrm{D}_{Unit} = \{\#\}_\perp$
$\mathrm{D}_{\tau\to\sigma} = (\mathrm{D}_\tau \to \mathrm{D}_\sigma)_\perp$
$\mathrm{Env}_N = \mathrm{Var} \to \mathrm{Val}_N$

Operations:

$\mathrm{Lift} : \mathrm{Val}_N \to (\mathrm{D}_{Unit} \to \mathrm{Val}_N)_\perp$
$\mathrm{Lift} = \lambda \mathrm{u}.\lambda \mathrm{v}.\mathrm{u}$

Valuation function:

$\mathbf{E} : \mathrm{Exp} \to \mathrm{Env} \to \mathrm{Val}_N$

$$\begin{aligned}
\mathbf{E}[\![x]\!] &= \underline{\lambda}\rho.\rho[\![x]\!] \\
\mathbf{E}[\![\mathtt{\#}]\!] &= \underline{\lambda}\rho.\# \\
\mathbf{E}[\![e_\tau\ e'_\sigma]\!] &= \underline{\lambda}\rho.(\mathbf{E}[\![e_\tau]\!]\rho)\ (\mathbf{E}[\![e'_\sigma]\!]\rho) \\
\mathbf{E}[\![\lambda x_\tau.e_\sigma]\!] &= \underline{\lambda}\rho.\lambda \mathrm{v}.\mathbf{E}[\![e_\sigma]\!][x \mapsto \mathrm{v}]\rho \\
\mathbf{E}[\![\mathtt{rec}\ x_\tau\ e_\tau]\!] &= \underline{\lambda}\rho.\mathbf{fix}\ (\lambda \mathrm{v}.\mathbf{E}[\![e_\tau]\!][x \mapsto \mathrm{v}]\rho)
\end{aligned}$$

Figure 7: Semantics of the non-strict typed lambda calculus

Translation of expressions:

$$\begin{aligned}
\mathbf{T}[\![x]\!] &= x\ \mathtt{\#} \\
\mathbf{T}[\![\mathtt{\#}]\!] &= \mathtt{\#} \\
\mathbf{T}[\![e_{\tau\to\sigma}\ e'_\tau]\!] &= \mathbf{T}[\![e_{\tau\to\sigma}]\!]\ (\underline{\lambda}\alpha.\mathbf{T}[\![e'_\tau]\!]) \\
\mathbf{T}[\![\lambda x_\tau.e_\sigma]\!] &= \underline{\lambda} x_{Unit\to\tau'}.\mathbf{T}[\![e_\sigma]\!] && \tau' = \mathbf{T}[\![\tau]\!] \\
\mathbf{T}[\![\mathtt{rec}\ x_\tau\ e_\tau]\!] &= \mathtt{rec}\ x_{\tau'}\ ((\underline{\lambda} x_{Unit\to\tau'}.\mathbf{T}[\![e_\tau]\!])\ (\underline{\lambda}\alpha_{Unit}.x)) && \tau' = \mathbf{T}[\![\tau]\!]
\end{aligned}$$

Translation of types:

$$\begin{aligned}
\mathbf{T}[\![\mathrm{Unit}]\!] &= \mathrm{Unit} \\
\mathbf{T}[\![\tau \to \sigma]\!] &= (\mathrm{Unit} \to \mathbf{T}[\![\tau]\!]) \to \mathbf{T}[\![\sigma]\!]
\end{aligned}$$

Figure 8: Translation

Now we are ready to show the translation. This can be seen in Figure 8. We give both a translation of expressions and a translation of types. This is necessary since a translated expression will in general have a different type than the original expression. We use $\mathbf{T}$ to denote both translations, since it will always be clear from the context what is being translated. In the translation we need to introduce new variables for the suspensions and this is usually an inconvenience, but in our case we can assume that the set of variables for the call by value language is equal to that of the call by name language extended with one new distinct variable α. This new variable can then be used for all occurrences of new variables in the translation. It can be shown that if an expression e is well typed and has type τ then $\mathbf{T}[\![e]\!]$ will be well typed

and have type $\mathbf{T}[\![\tau]\!]$. We will not show the proof of this here, since this is rather straightforward and tedious, but it can be found in Appendix A.1.

We will now prove that the translation in Figure 8 is indeed correct. This means that the meaning assigned to a call by name expression e by the call by name semantics and the meaning assigned to $\mathbf{T}[\![e]\!]$ by the call by value semantics are equivalent. Equivalence here means observable equivalence. That is, two constants are equivalent if they are the same constant or two function expressions e_τ and $e'_{\tau'}$ if for all equivalent arguments a_σ and $a'_{\sigma'}$: $(e_\tau\ a_\sigma)$ is equivalent to $(e'_{\tau'}\ (\lambda\alpha.a'_{\sigma'}))$. That $a'_{\sigma'}$ needs to be suspended here will be apparent when we start to formalize the notion of equivalence and presents the proof of correctness of the translation, but just looking at types would also suggest this.

We are now ready to define what we mean by equivalence:

Definition 5.1 We define a family of relations $\equiv_\tau$ on $\mathrm{D}_\tau \times \mathrm{D}_\sigma$ where $\sigma = \mathbf{T}[\![\tau]\!]$ by:

1. $\mathrm{v} \equiv_{Unit} \mathrm{w}$ **iff** $\mathrm{v} = \mathrm{w}$
2. $\mathrm{v} \equiv_{\tau'\to\tau''} \mathrm{w}$ **iff** $\forall\ \mathrm{u} \in \mathrm{D}_{\tau'}, \forall\ \mathrm{r} \in \mathrm{D}_{\sigma'}$: $\sigma' = \mathrm{T}[\![\tau']\!] \wedge \mathrm{u} \equiv_{\tau'} \mathrm{r} \Rightarrow \mathrm{v\ u} \equiv_{\tau''} \mathrm{w\ (Lift\ r)}$

It turns out that we need $\equiv_\tau$ to be inclusive, since we will be using fixed point induction in our proof, so we start by giving the definition of an inclusive predicate (taken from Schmidt [Schmidt 1986]):

Definition 5.2 A predicate $\mathrm{P{:}D} \to \mathit{Bool}$ is inclusive iff for every chain $\mathrm{C} \subseteq \mathrm{D}$, if $\mathrm{P(c)} = \mathrm{true}$ for every $\mathrm{c} \in \mathrm{C}$, then $\mathrm{P}(\bigsqcup\mathrm{C}) = \mathrm{true}$ also.

The following proposition is very similar to Proposition 3.13 of Søndergaard [Søndergaard 1989] and the proof is more or less the same. It shows that $\equiv_\tau$ is inclusive:

Proposition 5.1 $\equiv_\tau$ is inclusive for all $\tau \in \mathrm{Type}$.

Proof: By structural induction over types. The case $\tau = \mathrm{Unit}$ is trivial. Assume $\tau = \tau' \to \tau''$ and let $\mathrm{X} \subseteq \mathrm{D}_\tau \times \mathrm{D}_\sigma$ be a chain where $\mathrm{x}_1 \equiv_\tau \mathrm{x}_2$ holds for all $(\mathrm{x}_1, \mathrm{x}_2) \in \mathrm{X}$ and $\sigma = \mathrm{T}[\![\tau]\!]$. Let $\mathrm{u} \in \mathrm{D}_{\tau'}$ and $\mathrm{v} \in \mathrm{D}_{\sigma'}$ where $\sigma' = \mathrm{T}[\![\tau']\!]$ be given. Then $\mathrm{Y} = \{(\mathrm{x}_1\ \mathrm{u}, \mathrm{x}_2\ (\mathrm{Lift\ v})) \mid (\mathrm{x}_1, \mathrm{x}_2) \in \mathrm{X}\}$ is a chain in $\mathrm{D}_{\tau''} \times \mathrm{D}_{\sigma''}$ where $\sigma'' = \mathrm{T}[\![\tau'']\!]$. Assume further that $\mathrm{u} \equiv_{\tau'} \mathrm{v}$, we then have that $\mathrm{y}_1 \equiv_{\tau''} \mathrm{y}_2$ for all $(\mathrm{y}_1, \mathrm{y}_2) \in \mathrm{Y}$ and since $\equiv_{\tau''}$ is inclusive (by induction) $\mathrm{y}_1 \equiv_{\tau''} \mathrm{y}_2$ where $(\mathrm{y}_1, \mathrm{y}_2) = \bigsqcup\mathrm{Y}$. This means, since u and v were arbitrary, that:

$\forall\ \mathrm{u} \in \mathrm{D}_{\tau'}, \forall\ \mathrm{v} \in \mathrm{D}_{\sigma'}$: $\sigma' = \mathrm{T}[\![\tau']\!] \wedge \mathrm{u} \equiv_{\tau'} \mathrm{v} \Rightarrow (\bigsqcup\mathrm{X}_1)\ \mathrm{u} \equiv_{\tau''} (\bigsqcup\mathrm{X}_2)\ (\mathrm{Lift\ v})$

where $\bigsqcup\mathrm{X} = (\bigsqcup\mathrm{X}_1, \bigsqcup\mathrm{X}_2)$. This means that $\bigsqcup\mathrm{X}_1 \equiv_\tau \bigsqcup\mathrm{X}_2$. Hence $\equiv_\tau$ is inclusive. □

We extend $\equiv_\tau$ to work on environments in the following way ($\mathrm{Dom}(\rho)$ is the domain of ρ, in the sense that $\forall\ v{\in}\ \mathrm{Dom}(\rho)$: $\rho(v) \neq \bot$):

Definition 5.3 Let $\rho_N \in \mathrm{Env}_N$ and $\rho_S \in \mathrm{Env}_S$, then $\rho_N \equiv \rho_S$ if the following conditions hold:

1. $\mathrm{Dom}(\rho_N) = \mathrm{Dom}(\rho_S)$
2. $\forall\ x \in \mathrm{Dom}(\rho_N)$: $\rho_N\ x \equiv_\tau \rho_S\ x\ \#$

We are now ready to prove the correctness of the translation:

Theorem 5.1 Let $e_\tau \in \mathrm{Exp}$, $\rho_N \in \mathrm{Env}_N$ and $\rho_S \in \mathrm{Env}_S$, then

$\rho_N \equiv \rho_S \Rightarrow \mathbf{E}_N[\![e_\tau]\!]\rho_N \equiv_\tau \mathbf{E}_S[\![\mathbf{T}[\![e_\tau]\!]]\!]\rho_S$

Proof: The proof is by structural induction over $e_\tau \in \text{Exp}$. We assume that $\rho_N \equiv \rho_S$ and prove that $\mathbf{E}_N[\![e_\tau]\!]\rho_N \equiv_\tau \mathbf{E}_S[\![\mathbf{T}[\![e_\tau]\!]]\!]\rho_S$.

Case: x

$\mathbf{E}_N[\![x]\!]\rho_N =$
$\rho_N[\![x]\!] \equiv_\tau \rho_S[\![x]\!]\ \# =$ (since $\rho_N \equiv \rho_S$)
$\mathbf{E}_S[\![\mathbf{T}[\![x]\!]]\!]\rho_S$

Case: `#`

$\mathbf{E}_N[\![\texttt{\#}]\!]\rho_N = \# \equiv_{Unit} \# = \mathbf{E}_S[\![\mathbf{T}[\![\texttt{\#}]\!]]\!]\rho_S$

Case: $\lambda x_{\tau'}.e_{\tau''}$.

Let $\text{v} \in \text{D}_{\tau'}$ and $\text{w} \in \text{D}_{\sigma'}$ where $\sigma' = \mathbf{T}[\![\tau']\!]$ and assume that $\text{v} \equiv_{\tau'} \text{w}$. Then to prove this case of the theorem we have to prove:

$\mathbf{E}_N[\![\lambda x_{\tau'}.e_{\tau''}]\!]\rho_N\ \text{v} \equiv_{\tau''} \mathbf{E}_S[\![\underline{\lambda} x_{\sigma'}.\mathbf{T}[\![e_{\tau''}]\!]]\!]\rho_S\ (\text{Lift w})$

This is equivalent to proving:

$\mathbf{E}_N[\![e_{\tau''}]\!][x \mapsto \text{v}]\rho_N \equiv_{\tau''} \mathbf{E}_S[\![\mathbf{T}[\![e_{\tau''}]\!]]\!][x \mapsto (\text{Lift w})]\rho_S$

which follows from the induction hypothesis and equivalence of the two extended environments.

Case: $e_{\tau'\to\tau''}\ e'_{\tau'}$.

$\mathbf{E}_N[\![e_{\tau'\to\tau''}\ e'_{\tau'}]\!]\rho_N =$
$(\mathbf{E}_N[\![e_{\tau'\to\tau''}]\!]\rho_N)\ (\mathbf{E}_N[\![e'_{\tau'}]\!]\rho_N) \equiv_{\tau''}$ (by definition of $\equiv_\tau$ and induction hyp.)
$(\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'\to\tau''}]\!]]\!]\rho_S)\ (\text{Lift}\ (\mathbf{E}_S[\![\mathbf{T}[\![e'_{\tau'}]\!]]\!]\rho_S)) =$
$(\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'\to\tau''}]\!]]\!]\rho_S)\ (\mathbf{E}_S[\![\underline{\lambda}\alpha.\mathbf{T}[\![e'_{\tau'}]\!]]\!]\rho_S) =$
$\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'\to\tau''}]\!]\ (\underline{\lambda}\alpha.\mathbf{T}[\![e'_{\tau'}]\!])]\!]\rho_S =$
$\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'\to\tau''}\ e'_{\tau'}]\!]]\!]\rho_S$

Case: `rec` $x_{\tau'}\ e_{\tau'}$.

First we rewrite the left hand side for this case a little:

$\mathbf{E}_S[\![\mathbf{T}[\![\texttt{rec}\ x_{\tau'}\ e_{\tau'}]\!]]\!]\rho_S =$
$\mathbf{E}_S[\![\texttt{rec}\ x_{\tau''}\ ((\underline{\lambda} x_{Unit\to\tau''}.\mathbf{T}[\![e_{\tau'}]\!])\ (\underline{\lambda}\alpha.x))]\!]\rho_S =$
$\mathbf{fix}\ (\lambda \text{v}.\mathbf{E}_S[\![(\underline{\lambda} x_{Unit\to\tau''}.\mathbf{T}[\![e_{\tau'}]\!])\ (\underline{\lambda}\alpha.x)]\!][x \mapsto \text{v}]\rho_S) =$
$\mathbf{fix}\ (\lambda \text{v}.(\underline{\lambda} \text{v}.\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'}]\!]]\!][x \mapsto \text{v}]\rho_S)\ (\text{Lift v})) =$
$\mathbf{fix}\ (\lambda \text{v}.\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'}]\!]]\!][x \mapsto (\text{Lift v})]\rho_S)$

Since the right hand side is:

$\mathbf{E}_N[\![\texttt{rec}\ x_{\tau'}\ e_{\tau'}]\!]\rho_N = \mathbf{fix}\ (\lambda \text{v}.\mathbf{E}_N[\![e_{\tau'}]\!][x \mapsto \text{v}]\rho_N)$

what we have to prove is that:

$\mathbf{fix}\ (\lambda \text{v}.\mathbf{E}_N[\![e_{\tau'}]\!][x \mapsto \text{v}]\rho_N) \equiv_{\tau'} \mathbf{fix}\ (\lambda \text{v}.\mathbf{E}_S[\![\mathbf{T}[\![e_{\tau'}]\!]]\!][x \mapsto (\text{Lift v})]\rho_S)$

To prove this we prove that for $\text{F} \in \text{D}_{\tau'\to\tau'}$ and $\text{G} \in \text{D}_{\tau''\to\tau''}$, where $\tau'' = \mathbf{T}[\![\tau']\!]$, it holds that

$(\forall\ \text{v} \in \text{D}_{\tau'}, \forall\ \text{w} \in \text{D}_{\tau''}\colon\ \text{v} \equiv_{\tau'} \text{w} \Rightarrow \text{F v} \equiv_{\tau'} \text{G w}) \Rightarrow \mathbf{fix}\ \text{F} \equiv_{\tau'} \mathbf{fix}\ \text{G}$

This is easily proven by fixed point induction, since $\equiv_\tau$ is inclusive. This concludes the proof of the theorem. □

5.2 Strict lambdas

Until now the lambdas of our source language have been non-strict. We now discuss how to extend the translation to handle also strict lambdas. For the rest of the chapter we drop the types on expressions, since we really only needed these for our proof of Theorem 1. One way that we could translate the strict lambdas is by:

$$\mathbf{T}[\![\underline{\lambda}x.e]\!] = \underline{\lambda}x.(\underline{\lambda}x.(\underline{\lambda}x.\mathbf{T}[\![e]\!])\ (\underline{\lambda}\alpha.x))\ (x\ \texttt{\#})$$

and it would fit right into our translation and proof so far. Intuitively the rule expresses that we have to force the argument (by applying it to `#`) and then suspend the evaluated argument again (by wrapping it up in a lambda: $(\underline{\lambda}\alpha.x)$), before applying the strict lambda to it. On the other hand might seem that we would be doing a lot of work this way, when we are in fact translating into a language with only strict lambdas. A first simple optimization would be not to suspend the argument again. But to this we have to be able to distinguish variables bound by strict lambdas from variable bound by non-strict lambdas in our source language. We do this by underlining variable bound by strict lambdas. Now variable bound by strict lambdas will not have to be forced. The translation of `rec`-expressions can also be improved. If we look at the first translation shown in Figure 8, then we had to suspend the `rec`-bound variable in order to get the right result when the variable was used (i.e. all variables were forced before they were used). Since the evaluation of a variable always terminates, nothing is gained by first suspending it and then forcing it again. So we also underline `rec`-bound variables and remove the suspending of `rec`-bound variables in the translated `rec`-expressions. Introducing these three changes we get the new translation shown in Figure 9.

Translation of expressions:

$$\mathbf{T}[\![x]\!] = x\ \texttt{\#}$$
$$\mathbf{T}[\![\underline{x}]\!] = x$$
$$\mathbf{T}[\![\texttt{\#}]\!] = \texttt{\#}$$
$$\mathbf{T}[\![e\ e']\!] = \mathbf{T}[\![e]\!]\ (\lambda\alpha.\mathbf{T}[\![e']\!])$$
$$\mathbf{T}[\![\lambda x.e]\!] = \lambda x.\mathbf{T}[\![e]\!]$$
$$\mathbf{T}[\![\underline{\lambda}\,\underline{x}.e]\!] = \lambda x.(\lambda x.\mathbf{T}[\![e]\!])\ (x\ \texttt{\#})$$
$$\mathbf{T}[\![\texttt{rec}\ \underline{x}\ e]\!] = \texttt{rec}\ x\ \mathbf{T}[\![e]\!]$$

Figure 9: Translation with strict lambdas

It might still seem inefficient not to translate strict lambdas directly into strict lambdas, but this is however not always possible, since it may happen that strict and non-strict lambdas can be applied at the same application. This means that we cannot, when we translate an application $(e\ e')$, know whether all the lambdas that e can evaluate to will be strict or not.

This does on the other hand suggest a possible optimization of the translation. If by analyzing a program we can detect that only strict abstraction can be applied at an application $(e\ e')$, we can make it strict, that is, translate it into:

$$\mathbf{T}[\![e]\!]\ \mathbf{T}[\![e']\!]$$

Similarly all lambda abstractions $\underline{\lambda}x.e$ that we know only can be applied at strict applications, we can translate into:

$$\underline{\lambda}x.\mathbf{T}[\![e]\!]$$

That is we do not have to force the argument before applying the abstraction. But, we are still in trouble if strict and non-strict abstractions can be applied at the same applications. In the next section we will show a different type of translation that can handle the optimizations discussed and also the problem with strict and non-strict abstractions getting mixed up.

5.3 A Type Inference Based Translation

In this section we will outline a type inference based algorithm that will give a better translation than the one presented so far. The problem with the translation obtained so far is that all application are handled in the same way regardless what kind of abstraction may be applied at the application. If only strict abstraction can be applied at an application we still suspend the argument. This also result in that all strict abstractions are handled in the same way, since these are always applied to suspended arguments. The translation that we present in this section solves this problem.

Lets call applications, where only strict abstraction can be applied, strict applications. If we are going to translate strict applications in an efficient way we need to find these when given an expression to translate. One way to do this is to use a simplified closure analysis where abstract closures can be two things: either a strict (abstraction of strict abstractions) of non-strict (abstraction of non-strict abstractions). This would be a fairly straightforward analysis. But we also need to know which strict abstractions can only be applied at strict applications. So for this we would need a full closure analysis where abstract closures are labels identifying the lambda abstractions that they abstract (this is the same way Similix's closure analysis abstract closures [Bondorf 1990a]). It turns out that we can use a type inference based algorithm instead and we will now explain this.

The language that we translate is now even simpler than before. It contains base values c, variables x, strict and non-strict abstractions, i.e $\underline{\lambda}x.e$ and $\lambda x.e$, and applications $e_1\ e_2$, i.e. we have left out rec-expressions. We use this simple language to communicate the idea, but an extension of the idea to a language close to our definition language can be found in Appendix B. The syntax of the language and of the types we will be using can be seen in Figure 10.

Abstract syntax:

$e \in \text{Exp}$, $x \in \text{Var}$

$e ::= x \mid c \mid \lambda x.e \mid \underline{\lambda}x.e \mid e\ e$

$\tau \in \text{Type}$, $\alpha \in \text{IType}$

$\tau ::= \text{v}(\alpha) \mid \text{s}(\alpha)$
$\alpha ::= \text{Base} \mid \tau \rightarrow \tau$

Figure 10: Syntax of lambda calculus and types

The types have the following intuitive meaning. A type $\text{s}(\alpha)$ represents a suspended value, while $\text{v}(\alpha)$ represents a value either in *weak head normal form* or undefined. The inner types α are the "usual" type of the values, i.e. Base or function type.

We will start by presenting a type-checking system for expressions with *coercions*. If we look at the translation in Figure 9 then this corresponds to making all lambdas strict and inserting suspend and force operations at the right places. We can regard these operations as explicit coercions and if we insert these in a program we may use the type-checking to check whether these have been inserted in a consistent way. Our goal is of course to infer the coercions. Figure 11 shows the type inference system and coercion relations.

Type rules:

(CONST) $\Gamma \vdash c\colon \mathrm{v}(\mathrm{Base})$

(VAR) $$\frac{\Gamma(x) = \tau_x \quad \mathsf{c} : \tau_x \geq_{\downarrow} \tau}{\Gamma \vdash \mathsf{c}x\colon \tau}$$

(ABS) $$\frac{[x \mapsto \mathrm{v}(\alpha_x)]\Gamma \vdash e\colon \mathrm{v}(\alpha)}{\Gamma \vdash \underline{\lambda}x.e\colon \mathrm{v}(\mathrm{v}(\alpha_x) \to \mathrm{v}(\alpha))}$$

(ABS) $$\frac{[x \mapsto \mathrm{s}(\alpha_x)]\Gamma \vdash e\colon \mathrm{v}(\alpha)}{\Gamma \vdash \lambda x.e\colon \mathrm{v}(\mathrm{s}(\alpha_x) \to \mathrm{v}(\alpha))}$$

(APP) $$\frac{\Gamma \vdash e\colon \mathrm{v}(\tau \to \mathrm{v}(\alpha)) \quad \Gamma \vdash e'\colon \tau' \quad \mathsf{c}_1 : \tau' \leq_{\uparrow} \tau'' \quad \mathsf{c}_2 : \tau'' \leq_{\Uparrow} \tau}{\Gamma \vdash e \;\; \mathsf{c}_1\mathsf{c}_2e'\colon \mathrm{v}(\alpha)}$$

Coercion relations:

$\mathrm{nop} : \tau \geq_{\downarrow} \tau$ $\qquad$ $\mathrm{nop} : \tau \leq_{\uparrow} \tau$

$\downarrow \;\; : \mathrm{s}(\alpha) \geq_{\downarrow} \mathrm{v}(\alpha)$ $\qquad$ $\uparrow \;\; : \mathrm{v}(\alpha) \leq_{\uparrow} \mathrm{s}(\alpha)$

$\mathrm{nop} : \tau \leq_{\Uparrow} \tau$

$\Uparrow \;\; : t(\mathrm{v}(\alpha_1) \to \mathrm{v}(\alpha_2)) \leq_{\Uparrow} t(\mathrm{s}(\alpha_1) \to \mathrm{v}(\alpha_2)) \quad t = \mathrm{s}$ **or** $t = \mathrm{v}$

Figure 11: Rules for type-checking expressions with coercions

There are three explicit coercions. One corresponding to force, written $\downarrow$, that coerces a suspended value to a value; one corresponding to suspend, written $\uparrow$, that coerces a value to a suspended value; and a new one, written $\Uparrow$, that coerces a value-function (one whose argument is a value) to a suspend-function (one whose argument is suspended). If we look at Figure 9 again then we will see that all these three coercions are in fact implicitly present there. It is the new coercion $\Uparrow$ that makes the "mixing" of strict and non-strict function possible. The explicit coercions can be defined as in Figure 12.

$\uparrow(e) = (\underline{\lambda}\alpha.e)$
$\downarrow(e) = e$ #
$\Uparrow(e) = (\underline{\lambda}\nu.e\ (\nu$ #$))$

Figure 12: The Explicit Coercions

In this figure ν is a new variable introduced for the same reasons as α. We could have used α again, but using a new variable makes it clearer that ν and α do not serve the same purpose. As the definitions in Figure 12 is written, one should think that the coercions could be applied to any expression, but our system will limit their use to certain kinds of expressions. The coercions can be placed at exactly the same places as the translation in Figure 9 may place them, except that $\Uparrow$ may be place in front of any argument to an application. The improvement is that we can now leave out the coercions and place $\Uparrow$ at other places than in front of the abstraction.

The coercion relations $\_ :\_ \geq_{\downarrow} \_ \subseteq$ Coercion $\times$ Type $\times$ Type etc. relates the coercions to the change in type caused by the coercions. If the type-checking of an expression with coercions succeeds, then we can safely replace the coercions by the above definitions and make all lambdas strict.

Let us look at an example. Consider the expression:

$(\underline{\lambda}\text{x.x (x }(\lambda\text{u.u}))) (\underline{\lambda}\text{z.z }(\underline{\lambda}\text{v.v}))$

Here the variable z will be bound to both a strict and a non-strict lambda. If we use our original translation (Figure 9) on this expression we would get a result equivalent to:

$(\underline{\lambda}\text{x.x} \uparrow(\text{x} \uparrow(\underline{\lambda}\text{u.}\downarrow\text{u}))) \uparrow\Uparrow(\underline{\lambda}\text{z.z} \uparrow\Uparrow(\underline{\lambda}\text{v.v}))$

The coercions are not explicitly present in Figure 9, but if we insert the explicit coercions in this expression we get an expression that is equal to the one that the translation would yield, except for renaming of variables. Now, the following much better result is valid according to the coercion checking system:

$(\underline{\lambda}\text{x.x} \Uparrow(\text{x } (\underline{\lambda}\text{u.}\downarrow\text{u}))) (\underline{\lambda}\text{z.z} \uparrow(\underline{\lambda}\text{v.v}))$

and the translated expression of this is:

$(\underline{\lambda}\text{x.x } (\underline{\lambda}\nu\text{.x } (\underline{\lambda}\text{u.u \#}) (\nu\ \#))) (\underline{\lambda}\text{z.z } (\underline{\lambda}\alpha.\underline{\lambda}\text{v.v}))$

This shows that we may get much better result if we could infer the coercions than just always inserting them as the translation original suggests. Especially in cases where strict and non-strict functions are completely separated, the method would give very good results. So let us see if and how we could infer the coercions.

A simple way to infer types and explicit coercions could be to apply some suitable variant of the W-algorithm [Damas and Milner 1982] for all possible insertions of coercions and then choose the best one of these in some sense. This is related to what Gomard did in his binding time analysis for the lambda-mix partial evaluator [Gomard 1989]. Gomard had his version of the W-algorithm return some information on where to lift expressions from static to dynamic (code). His binding time analysis algorithm then used this information in its next iteration. Gomard's work has been extended by Henglein [Henglein 1991] and here we will outline how we can use techniques similar to Henglein's on our system.

Instead of finding a proof with our type checking system we find a solution to an equivalent constraint system on type expressions. Since the types inference rules are syntax-directed, i.e. there is only one rule for each syntactical form, the proof tree for each lambda expression can only have one structure. The only thing that may vary is the choice of coercions. This suggest that the structure of the proof is not important and we shall indeed show by an example how we can change the rules of our type system in such a way that we may forget the the structure of the proof tree and work only with sets of constrains.

Consider the APP rule of Figure 11. We now replace all types in this rule, that are not type variables, by new type variable and add new constraints to the rule stating the equivalence between the types and the variable that replace them. We get the equivalent rule:

$$\frac{\begin{array}{l}\Gamma \vdash e\colon \tau'_e \quad \Gamma \vdash e''\colon \tau_{e''} \\ \tau_{e'} = \mathrm{v}(\overline{\overline{\tau}}_{e''} \to \tau_e) \quad \tau_e = \mathrm{v}(\alpha_e) \quad \mathsf{c}_{e''}{:}\tau_{e''} \leq_{\uparrow} \overline{\tau}_{e''} \quad \overline{\mathsf{c}}_{e''}{:}\overline{\tau}_{e''} \leq_{\Uparrow} \overline{\overline{\tau}}_{e''}\end{array}}{\Gamma \vdash e'\ \mathsf{c}_{e''}\overline{\mathsf{c}}_{e''}e''\colon \tau_e} \qquad (e = e'\ \mathsf{c}_{e''}\overline{\mathsf{c}}_{e''}e'')$$

The type variable with bars over are new type variables and the variables $\mathsf{c}_{e''}$ etc. range over coercions (nop, $\uparrow$, etc.). From this we can now read the generated constraints directly. If we assume that expressions are syntactically well-formed then all that has to be satisfied by the rule are the constraints and we can by doing the same to all the other rules obtain a complete set

$$
\begin{array}{l}
\mathrm{C}(e) = \\
\quad \textbf{cases } e \textbf{ of} \\
\qquad \begin{array}{ll}
c & : \{\tau_e = \mathrm{v}(\mathrm{Base})\} \\
x & : \{\mathrm{c}_e{:}\tau_x \geq_{\downarrow} \tau_e\} \\
\underline{\lambda} x.e' & : \{\tau_e = \mathrm{v}(\tau_x \rightarrow \tau_{e'}),\ \tau_x = \mathrm{v}(\alpha_x),\ \tau_{e'} = \mathrm{v}(\alpha_{e'})\} \bigcup \mathrm{C}(e') \\
\lambda x.e' & : \{\tau_e = \mathrm{v}(\tau_x \rightarrow \tau_{e'}),\ \tau_x = \mathrm{s}(\alpha_x),\ \tau_{e'} = \mathrm{v}(\alpha_{e'})\} \bigcup \mathrm{C}(e') \\
e'\ e'' & : \{\tau_{e'} = \mathrm{v}(\overline{\overline{\tau}}_{e''} \rightarrow \tau_e),\ \tau_e = \mathrm{v}(\alpha_e),\ \mathrm{c}_{e''}{:}\tau_{e''} \leq_{\uparrow} \overline{\tau}_{e''},\ \overline{\mathrm{c}}_{e''}{:}\overline{\tau}_{e''} \leq_{\Uparrow} \overline{\overline{\tau}}_{e''}\} \bigcup \\
& \quad \mathrm{C}(e') \bigcup \mathrm{C}(e'')
\end{array}
\end{array}
$$

Figure 13: Translation into constraint system

of constrains that the type of a given expressions must satisfy in order to be well-typed. Figure 13 shows the complete translation of expressions into sets of constraints.

It still remains to be shown that the method is correct, how to normalize the constraint system and whether there is a minimal solution (with respect to number of coercions). That a solution exist is trivial, since the one generated by the translation in Figure 9 is a solution. We believe that a unique minimal solution can be found. A problem is the that there may be different solutions with the same number of coercions that only differs in the position of the $\Uparrow$ coercions. However, we can make the solution unique, by imposing the extra restriction, that the $\Uparrow$ coercions should be placed as late as possible according to some order. The coercion variables c in the constrains can be viewed as a kind of pointers back to the position in the the original expression where an explicit coercion may be inserted. The variables should then be instantiated during normalization of the constraint system. We will leave it to future work to show these things formally.

An other point of interest is to extend the language with conditionals, case expressions etc. Appendix B shows a suggestion for such an extension of the type system. This is certainly of interest if we are going to translate a denotational definition language into a strict language. We believe it to be fairly easy to extend the method to handle all of the definition language in Figure 3.

5.4 Translation of the Denotational Language

Until now, the language that our translation can treat has been quite simple, but as we discussed in the last section it should be possible to extend it to treat all of the denotational definition language defined in Section 3. In order to translate a denotational definition we first would have to infer the explicit coercions and insert these in the definition. This means that the underlining of lambda is something that have to be given from the start. These could all have been put there by hand or some of them could have been found by analyzing the definitions. There are several ways in which we may discover lambdas that can be underlined. Firstly, lambdas that are applied to syntax argument can be underlined. Secondly, strictness analysis may discover some case in which lambdas could be underlined, but might overlook some obvious cases, namely lambdas binding environment variables. Often there are cases in a semantics where the environment argument is not used (constant expressions), and these would prevent us from making a given function strict in an environment argument. Finally, we may use termination analysis to detect even more cases, hopefully some of the important ones missed by the strictness analysis.

Then when the coercions have been found, we can replace these with the definition of the explicit coercions as shown in Figure 12.

The whole translation can then be regarded as a translation from our denotational definition language into itself, but where all lambdas in the target definition are strict except those bound

to **fix**. It is then fairly straightforward to translated the target definition into an interpreter written in a strict language. In Section 5.6 we discuss how this can be done using Scheme.

5.5 The Applicative Fixed Point Operator

We have now seen that we can translate a general denotational definition into a denotational definition with only strict lambdas. This means that we may implement it fairly direct in a strict language. We will discuss how to do so in Scheme in the next section. But before we do that there is a issue of general interest concerning implementation of the fixed point operator in a strict language.

If the implementation language has local recursive definitions we can simply translate **fix**($\lambda f.expr$) into a local recursive definition. In Scheme this would mean that we would translate **fix**($\lambda f.expr$) into:

```
(letrec ((f expr)) f)
```

where `expr` is the translation of *expr*. But this only holds for *expr* of function type in general, since Scheme will evaluate `expr` before the body of the letrec-expression. However, it does hold if evaluation of `expr` terminates. This will be a problem in all strict languages that does not have any non-strict constructors other than abstractions.

An other problem with this way of defining **fix** is that all self-applicable partial evaluators for functional languages existing today do not handle local recursive definitions. This also goes for Similix. So we have to find some other way to define **fix** in a strict functional language. We will show how to do so in Scheme. We can of course make make the definition above global, but the we would have to make all free variable of *expr* argument to the new definition and also make sure that no names of local definition clash. The method that we will describe here in fact make that the responsibility of the partial evaluator.

Fortunately we can define a function that behaves like **fix** in Scheme as long as the fixed point defined is a function. This is called the *applicative fixed point operator* and that uses the implicit fixed point operator in the Scheme semantics. It can be defined as:

```
(define (fix f) (lambda (x) ((f (fix f)) x)))
```

by using this we can for example define the factorial function by:

```
(define (fac x)
  (fix
    (lambda (f)
      (lambda (x)
        (if (= x 0) 1 (* x (f (- x 1))))))))
```

One can show that the above definition of `fix` does indeed give the right fixed point for functionals defining functions. A proof of this can be found in Appendix A.1.

5.6 Translating into Scheme

The translation presented so far has been kept as independent of the target language as possible. What is demanded of this is that its a strict functional language with abstractions. This means that languages like LISP, SML and Scheme are suitable candidates. Since we want to partially evaluate the interpreters with Similix it is natural that we take a look at the specific problems related with translation into Scheme. It turns out that Scheme provides us with a very simple way to treat the introduction of new variables that occurs in the $\uparrow$ coercion. The newly introduced

variable α is never used so it is only a kind of dummy variable. This means that in Scheme we can use `(lambda () ...)` for ($\lambda\alpha$. ...) and `(x)` for (x `#`). The $\Uparrow$ coercion must however be treated with care. If the coercion is placed immediately surrounding a lambda abstraction:

$\Uparrow(\underline{\lambda}x.\, expr)$

we may translate this into:

`(lambda (x) (let ((x (x)))`*expr'*`))`

where *expr'* is the translated expression of *expr*. Otherwise we have to be careful not to shadow any free in the expression being coerced. We may get around this problem by using a higher order function for the explicit coercion:

```
suspend-function = (lambda (f) (lambda (x) (f (x))))
```

This in not as efficient as is could be in some cases, but Similix will unfold the application of `suspend-function` to its argument and take care of the naming problem when specializing the interpreters.

An important issue that we have not touched upon in general is the representation of syntactical and semantic domains in the implementation language. In the case of Scheme the representation of syntactical domains is quite simple, any representation, using list or vectors to represent nodes in the abstract syntax, will suffice. An example will be given below.

Even the representation of semantic domains may be quite simple in Scheme. We may use some simple Scheme values for the values in basic domains, for example Scheme integers for integers, etc. For values in compound domains we can use vectors for product domain values, pairs where the first component is a tag, for sum domain values and Scheme functions for function values. This kind of representation is simple, but we may sometimes benefit from using Schemes own internal tags for sum domains to avoid doing needless tagging and untagging. We may use this method either when the language that we are implementing is statically typed and type checking has been performed before we interpret, or when we are ready to accept that Scheme's dynamic type-checking captures and report any type error of our source language.

Some Scheme implementations have pattern matching in the form of record-case-expressions, but since Similix cannot handle any form of pattern matching we need to consider how to translate the pattern matching of the semantics into Scheme. We will show how to do this by an example in the next section.

If the syntax of an expression has the form:

$expr \rightarrow$ ... | **if** $expr_1$ **then** $expr_2$ **else** $expr_3$ | ...

then we can represent an if-expression **if** $expr_1$ **then** $expr_2$ **else** $expr_3$ like by a Scheme list `(if` $expr'_1$ $expr'_2$ $expr'_3$ `)` where $expr'_i$ is the representation of $expr_i$. For this we will need a function `ExprIf?` that tests whether an expression is an if-expression or not and functions `ExprIf->expr1`, `ExprIf->expr2`, `ExprIf->expr3` that decompose an if-expression into its subexpressions. These could then be defined as:

```
(define (ExprIf? expr) (and (pair? expr) (equal? (car expr) 'if)))
(define ExprIf->expr1 cadr)
(define ExprIf->expr2 caddr)
(define ExprIf->expr3 cadddr)
```

Usually these functions will be defined a primitive operations when the interpreter is used with the Similix partial evaluator to generate a compiler. We can do this, since the operations will

always be used with static arguments, and there is therefore no need for the specializer to know their representation.

5.7 A Small Example

In this section we give an example of how to translate a language definition in our denotational definition language into an interpreter for the language written in Scheme. The defined language used is taken from Bondorf [Bondorf 1990a] and is a statically scoped lambda calculus extended with constants, conditionals, binary primitive operations and recursive "let". We will call the language Small for future reference. In [Bondorf 1990a] it is not discussed whether the language is typed or not, but it can easily be given a type system (eg. a small extension of the system in Section 5.1). The syntax and semantics of the language is given in Figure 14.

Abstract syntax:

$p \in \text{Prog}$, $e \in \text{Exp}$, $c \in \text{Const}$, $v \in \text{Var}$, $b \in \text{Binop}$

$p ::= e$

$e ::= c \mid v \mid \texttt{(}b\ e\ e\texttt{)} \mid \texttt{(if }e\ e\ e\texttt{)} \mid \texttt{(lambda }v\ e\texttt{)} \mid \texttt{(letrec }v\ e\ e\texttt{)} \mid \texttt{(}e\ e\texttt{)}$

Semantic algebra:

Domains:

$\mathbf{Var}$ = semantic representation of variables (omitted)
Base = basic values (omitted)

$\text{w} \in \text{Val} = (\text{Base} + (\text{Val} \to \text{Val}))_\perp$
$\rho \in \text{Env} = \mathbf{Var} \to \text{Val}$

Operations:

$[\_ \mapsto \_]\_ : \mathbf{Var} \to \text{Val} \to \text{Env} \to \text{Env}$
$[\text{v} \mapsto \text{w}]\rho = \underline{\lambda}\text{v}_1.\text{v}_1{=}\text{v} \to \downarrow\text{w} \,[]\, \rho\ \text{v}_1$

Valuation function:

$\mathbf{P} : \text{Prog} \to \text{Val} \to \text{Val}$
$\mathbf{P}\ [\![e]\!] = \underline{\lambda}\text{i}.\mathbf{E}[\![e]\!](\underline{\lambda}\text{w}.\uparrow\text{i})$

$\mathbf{E} : \text{Exp} \to \text{Env} \to \text{Val}$
$\mathbf{E}[\![v]\!] = \underline{\lambda}\rho.\rho(\mathbf{V}[\![v]\!])$
$\mathbf{E}[\![c]\!] = \underline{\lambda}\rho.\mathbf{C}[\![c]\!]$
$\mathbf{E}[\![\texttt{(}b\ e\ e'\texttt{)}]\!] = \underline{\lambda}\rho.\mathbf{B}[\![b]\!]\ (\mathbf{E}[\![e]\!]\rho)\ (\mathbf{E}[\![e']\!]\rho)$
$\mathbf{E}[\![\texttt{(if }e\ e'\ e''\texttt{)}]\!] = \underline{\lambda}\rho.\mathbf{E}[\![e]\!]\rho \to \mathbf{E}[\![e']\!]\rho \,[]\, \mathbf{E}[\![e'']\!]\rho$
$\mathbf{E}[\![\texttt{(lambda }v\ e\texttt{)}]\!] = \underline{\lambda}\rho.\lambda\text{w}.\mathbf{E}[\![e]\!][\mathbf{V}[\![v]\!] \mapsto \text{w}]\rho$
$\mathbf{E}[\![\texttt{(letrec }v\ e\ e'\texttt{)}]\!] = \underline{\lambda}\rho.\mathbf{E}[\![e']\!]\ \mathbf{fix}(\lambda\rho'.[\mathbf{V}[\![v]\!] \mapsto \uparrow(\mathbf{E}[\![e]\!]\rho')]\rho)$
$\mathbf{E}[\![(e\ e')]\!] = \underline{\lambda}\rho.(\mathbf{E}[\![e]\!]\rho)\ \uparrow(\mathbf{E}[\![e']\!]\rho)$

$\mathbf{V}$: Var $\to$ $\mathbf{Var}$ (omitted)
$\mathbf{C}$: Const $\to$ Val (omitted)
$\mathbf{B}$: Val $\to$ Val $\to$ Val (omitted)

Figure 14: Semantics of Small

Before applying our transformation to the definition we have to decide which lambdas can be made strict. The first and most natural lambdas to look at are those taking environments

as arguments. An environment can be either the initial environment or built by applying the update operation or **fix** to other environments. All these operations terminates so we can safely make all lambdas binding environments strict. Also the lambdas in the environments can be made strict, since these are always applied to syntax arguments. This also goes for two of the "hidden" lambdas that we may overlook by using the function update notation. The application $[\mathrm{v} \mapsto \mathrm{w}]\rho$ really means upd v w ρ, where

$$\mathrm{upd} = \underline{\lambda}\mathrm{v}.\lambda\mathrm{w}.\underline{\lambda}\rho.\underline{\lambda}\mathrm{v}_1.\mathrm{v}_1{=}\mathrm{v} \rightarrow \rho\ \mathrm{w} \,[]\, \rho\ \mathrm{v}_1$$

The lambdas corresponding to first and last formals of this definition can be made strict along with the lambda ($\lambda \mathrm{v}_1$. ...) in the right hand side of the definition. This gives the annotations of lambdas as strict that can be seen in Figure 14. We have also in Figure 14 shown where the coercions would be inserted by our type inference based method. When reading the semantics one may just disregard from these.

```
(extend-syntax (my-delay) ((my-delay v) (lambda () v)))
(define (my-force d) (d))
;
(loadt "scheme.adt")
(loadt "cbn-int.adt")
;
(define (run e w)
  (_E e (lambda (v) (my-delay w))))
;
; _E: Exp -> Env -> Val
(define (_E e r)
  (cond
   ((EVar? e) (r (EVar->Var e)))
   ((EConst? e) (_C (EConst->Const e)))
   ((EBinop? e) ((_B (EBinop->Binop e))
                 (_E (EBinop->E1 e) r)
                 (_E (EBinop->E2 e) r)))
   ((EIf? e) (if (_E (EIf->E1 e) r) (_E (EIf->E2 e) r) (_E (EIf->E3 e) r)))
   ((ELambda? e) (lambda (w)
                   (_E (ELambda->E e)
                       (upd (ELambda->Var e) w) r)))
   ((ELetrec? e) (_E (ELetrec->E2 e)
                     (fix
                       (lambda (r1)
                         (upd (ELetrec->Var e)
                              (my-delay (_E (ELetrec->E1 e) r1))
                              r)))))
   (else
    ((_E (EApply->E1 e) r) (my-delay (_E (EApply->E2 e) r))))))

(define (fix f) (lambda (x) ((f (fix f)) x)))

(define (upd v w r)
  (lambda (v1) (if (equal? v v1) (my-force w) (r v1))))
```

Figure 15: Scheme interpreter for Small

Figure 15 shows the interpreter that we would obtain by systematically applying our transformation to the semantics of Figure 14. The coercion corresponding to suspension is defined

as extended syntax (a macro) and is called `my-delay`. We have to define `my-delay` as extended syntax and not as a function, since function evaluate their arguments. The force coercion on the other hand is defined as a function and is called `my-force`. If we compare our interpreter with that of [Bondorf 1990a] we find that the two interpreters are almost identical. One difference is that in [Bondorf 1990a] the case for variables looks like:

```
((EVar? e) (my-force (r (EVar->Var e))))
```

The reason for this difference is that the force coercion in our `upd` function is moved out of `upd` and placed at the only place in the interpreter where the environment is applied. Besides this difference the two interpreters are identical.

This is a very promising result, because it suggests that it should be possible to automate the translation of denotational definitions into interpreters. We will return to the discussion of that in Chapter 9.

5.8 Summary

We conclude this chapter with a summary of the methods described. We assume that we start of with a denotational definition of a language written in the language of Chapter 4 and we want obtain an interpreter for the language written in a strict functional language. We choose Scheme for the sake of the example and now an interpreter can be obtained through the following step:

1. Apply type inference to the definition to obtain a new definition annotated with coercions. Functions are assumed strict in syntax arguments.

2. Replace the annotations with the explicit coercions and make all lambdas strict, except those that are directly connected to fixed point operators.

3. Compile the pattern matching of the definitions into equivalent code without pattern matching, using conditional-expressions only. Efficiency is not of vital importance here if one is concerned with compiler generation by partial evaluation, since it is only the efficiency of the generated compiler that is affected by this, not of the target programs.

4. Now the "definition" is in a form that can easily be translated into Scheme. For the sake of doing compiler generation using Similix we must add some suitable definition of the fixed point operator, as explained in Section 5.5, to the final interpreter. We also have to find representations in Scheme of our syntactical and semantic domains during this step.

This conclude our treatment of issues concerning the translation of denotational definitions into interpreters. In the next chapter we will discuss a way in which this translation may be automated.

Chapter 6

Meta-Interpreters

In Section 3.3.2 we discussed an alternative method to do compiler generation using partial evaluation. We will now shortly return to this method. It involves writing a meta-interpreter for some definition language and using this to generate interpreters for languages. Recall that a meta-interpreter is a program that take a language definition for some language L and an L-program P and runs P according to the definition. In this chapter we will present such a meta-interpreter for a denotational definition language. We will present it in a notation that could be viewed as a denotational semantics, but that we also know how to implement in a strict language remembering the results of Chapter 5. The treatment will be very informal and we will omit definitions of some domains, functions, etc., but we hope that it will become clear how we may formally derive meta-interpreters and that the structure of our meta-interpreter is quite natural.

6.1 A Meta-interpreter

The definition language that we will use is a subset of the one defined in chapter 4 after coercions have been added. This means that all lambdas are strict except the ones in `fix` (`fix` can now only be applied to lambda abstractions). Abstract syntax pattern now have a simple form: c { asv } where c is a node in the abstract syntax and asv is a variable ranging over abstract syntax. The syntax of the language can be seen in Figure 16.

Syntax variables:

def^* ∈ Definition*, alt^* ∈ Alternative*
asv^* ∈ AbstractSyntaxVariable* = Variable*
s^* ∈ AbstractSyntax*

Abstract syntax:

def	::= eq { eq }	(definition)
eq	::= $fun\ alt$ { alt }	(definition of valuation function)
alt	::= asp `=` $expr$	(alternative)
$expr$	::= $cst \mid var \mid op \mid expr \rightarrow expr \,[]\, expr \mid$ $expr\ expr \mid \underline{\lambda} var.expr \mid$ `fix` $(\lambda var.expr) \mid$ $fun \mid asv \mid \uparrow expr \mid \Uparrow expr \mid \downarrow expr$	
asp	::= c { asv }	
s	::= $c\ s^*$	(abstract syntax of the language being defined)

Figure 16: Syntax of the Definition language with Coercions

Now, recall that the way denotational semantics assigns meaning to programs is by translating the programs into a semantic language that we know how to interpret. Here this will essentially be some form of applied lambda calculus. We will call this translation **U** and Figure 17 shows the definition of **U**.

Substitutions:
$\delta \in$ ValuationFunctionName $\rightarrow \Lambda_a$ = Applied lambda calculus expressions
$\sigma \in$ AbstractSyntaxVariable $\rightarrow \Lambda_a$

Translation:
$\mathbf{U}[\![def^*]\!] = \mathbf{fix}\ (\lambda\delta.\lambda\mathrm{f}.\lambda\mathrm{s}.\mathbf{D}[\![def^*]\!]\delta\ \mathrm{f}\ \mathrm{s})$
$\mathbf{D}[\![(fun\ alt^*)\ def^*]\!]\delta\ \mathrm{f}\ \mathrm{s} = ([\![fun]\!] = \mathrm{f}) \rightarrow \mathbf{A}[\![alt^*]\!]\delta\ \mathrm{s}\ \square\ \mathbf{D}[\![def^*]\!]\delta\ \mathrm{f}\ \mathrm{s}$
$\mathbf{A}[\![(c\ asv^*\ expr)\ alt^*]\!]\delta[\![(c'\ s^*)]\!] =$
$([\![c]\!] = [\![c']\!]) \rightarrow \mathbf{U}_E[\![expr]\!]\delta[[\![asv^*]\!] \mapsto [\![s^*]\!]]\sigma_0\ \square\ \mathbf{A}[\![alt^*]\!]\delta[\![(c'\ s^*)]\!]$

Figure 17: Translation Defined by the Definition Language

The inputs to **U** are beside the definition *def**, the name f of a valuation function in *def** and a syntax phrase s of the language being defined. This means that $\mathbf{U}[\![def^*]\!]$ f translates syntax phrases of the kind that the valuation function f gives meaning to. The function $\mathbf{U}_E$ translate expressions and is nearly an identity transformation except for valuation functions and abstract syntax arguments where the definition is which it translate like:

$$\mathbf{U}_E[\![fun]\!]\delta\sigma = \delta[\![fun]\!]$$
$$\mathbf{U}_E[\![asv]\!]\delta\sigma = \sigma[\![asv]\!]$$

and explicit coercions which it translate according to the definitions in Figure 12.

If we have a call by value semantics for the applied lambda calculus with valuation function **E**, we can get a semantics with valuation function **M** for the definition language by:

$$\mathbf{M}[\![def]\!]\ \mathrm{f}\ \mathrm{s} = \mathbf{E}_\rho[\![\mathbf{U}[\![def]\!]\ \mathrm{f}\ \mathrm{s}]\!]$$

where the notation $\mathbf{E}_\rho[\![e]\!]$ means $\mathbf{E}[\![e]\!]\rho$. Having this definition of the semantics of the definition language we can try to simplify it to get a more direct definition of the semantics. First we fold $\mathbf{E}_\rho$ over **U**. The correctness of this step depends on two conditions. Firstly, both the definitions and the syntax phrases being defined must be syntactically well-formed. Secondly, the valuation function names used in a definition must be defined in the definition and the abstract syntax variables used in an equation must be bound in the abstract syntax pattern of the equation. In other words the translation define by the definition must always succeed. We can now combine $\mathbf{E}_\rho[\![\mathbf{U}_E[\![expr]\!]\delta\sigma]\!]$ into a call to one function $\mathbf{R}[\![expr]\!]\delta\sigma\rho$. The final result of doing all these simplifications can be seen in Figure 18, which shows the semantics of our definition language or simply a definition of our meta-interpreter.

This definition of the meta-interpreter is the directly implementable in a strict language like Scheme, since all lambdas, except the one bound to **fix**, are strict.

6.2 Experiments

We have conducted two simple experiments with an implementation of the meta-interpreter in Scheme, one for the functional language defined in Figure 15 of Section 5.7 and one for a

$$\mathbf{M}[\![\, def^* \,]\!] = \mathbf{fix}\ (\lambda\delta.\lambda\mathrm{f}.\lambda\mathrm{s}.\mathbf{D}[\![\, def^* \,]\!]\delta\ \mathrm{f}\ \mathrm{s})$$

$$\mathbf{D}[\![\, (fun\ alt^*)\ def^* \,]\!]\delta\ \mathrm{f}\ \mathrm{s} = ([\![fun]\!] = \mathrm{f}) \rightarrow \mathbf{A}[\![\, alt^* \,]\!]\delta\ \mathrm{s} \square \mathbf{D}[\![\, def^* \,]\!]\delta\ \mathrm{f}\ \mathrm{s}$$

$$\mathbf{A}[\![\, (c\ asv^*\ expr)\ alt^* \,]\!]\delta[\![\, (c'\ s^*) \,]\!] = ([\![c]\!] = [\![c']\!]) \rightarrow \mathbf{R}[\![\, expr \,]\!]\delta[[\![\, asv^* \,]\!] \mapsto [\![\, s^* \,]\!]]\sigma_0 \square \mathbf{A}[\![\, alt^* \,]\!]\delta[\![\, (c'\ s^*) \,]\!]$$

$$\mathbf{R}[\![\, \downarrow expr \,]\!]\delta\sigma\rho = \mathbf{R}[\![\, expr \,]\!]\delta\sigma\rho\ \#$$
$$\mathbf{R}[\![\, \uparrow expr \,]\!]\delta\sigma\rho = (\underline{\lambda}\alpha.\mathbf{R}[\![\, expr \,]\!]\delta\sigma\rho)$$
$$\mathbf{R}[\![\, \Uparrow expr \,]\!]\delta\sigma\rho = (\underline{\lambda}\alpha.\mathbf{R}[\![\, expr \,]\!]\delta\sigma\rho\ (\alpha\ \#))$$
$$\mathbf{R}[\![\, cst \,]\!]\delta\sigma\rho = \mathrm{C}[\![\, cst \,]\!]$$
$$\mathbf{R}[\![\, var \,]\!]\delta\sigma\rho = \rho[\![\, var \,]\!]$$
$$\mathbf{R}[\![\, op \,]\!]\delta\sigma\rho = \mathbf{O}[\![\, op \,]\!]$$
$$\mathbf{R}[\![\, expr_1 \rightarrow expr_2 \square\ expr_3 \,]\!]\delta\sigma\rho = \mathbf{R}[\![\, expr_1 \,]\!]\delta\sigma\rho \rightarrow \mathbf{R}[\![\, expr_2 \,]\!]\delta\sigma\rho \square \mathbf{R}[\![\, expr_3 \,]\!]\delta\sigma\rho$$
$$\mathbf{R}[\![\, \underline{\lambda} var.expr \,]\!]\delta\sigma\rho = \underline{\lambda}\mathrm{v}.\mathbf{R}[\![\, expr \,]\!]\delta\sigma[[\![\, var \,]\!] \mapsto \mathrm{v}]\rho$$
$$\mathbf{R}[\![\, e_1\ e_2 \,]\!]\delta\sigma\rho = \mathbf{R}[\![\, e_1 \,]\!]\delta\sigma\rho\ (\mathbf{R}[\![\, e_2 \,]\!]\delta\sigma\rho)$$
$$\mathbf{R}[\![\, \texttt{fix}\ (\lambda var.expr) \,]\!]\delta\sigma\rho = \mathbf{fix}\ (\lambda\mathrm{v}.\mathbf{R}[\![\, expr \,]\!]\delta\sigma[[\![\, var \,]\!] \mapsto \mathrm{v}]\rho)$$
$$\mathbf{R}[\![\, fun \,]\!]\delta\sigma\rho = \delta[\![\, fun \,]\!]$$
$$\mathbf{R}[\![\, asv \,]\!]\delta\sigma\rho = \sigma[\![\, asv \,]\!]$$

Figure 18: The Meta-interpreter

small imperative language. These both show promising results and especially that it is possible to generate interpreters with good binding time properties. The generated interpreter for the functional language is in fact very close in structure to the one in Figure 15.

The imperative language is a small language called Tiny. It was taken from Paulson [Paulson 1984], where also the semantics of the language was found. Figure 19 show this semantics.

6.2.1 Measuring run times

All the tests were run on Sparc station 2/Sun OS 4.1. The Scheme system used was Chez Scheme Version 3.2. The run times and the the storage usages were measured using the `time` function of Chez Scheme. The run times were found as the average of several runs to get a more accurate result.

6.2.2 Results

Figure 20 shows the run times for the example program. It shows the time it took to run the meta-interpreter on the both the definition and the program, the time it took to run the generated interpreter on the program and finally the time it took to run the target program. In all cases the input was 10. The program used in the experiment was the factorial program (here in concrete syntax):

```
(letrec fac (lambda x
              (if (= x 0)
                  1
                  (* x (fac (- x 1)))))
        (fac input))
```

Abstract syntax:
$p \in \text{Prog}$, $c \in \text{Com}$, $e \in \text{Exp}$, $id \in \text{Var}$, $n \in \text{Integer}$

$p ::= \text{Prog } c\ id$
$c ::= \text{Skip} \mid \text{Assign } id\ e \mid \text{Seq } c\ c \mid \text{If } e\ c\ c \mid \text{While } e\ c$
$e ::= \text{Ident } id \mid \text{Int } n \mid \text{Add } e\ e \mid \text{Sub } e\ e \mid \text{Mul } e\ e$

Semantic algebra:

Domains:
$\text{v} \in \text{Val} = \text{Integer}_\perp$
$\sigma \in \text{Store} = \text{Var} \to \text{Val}$

Operations:
update: $\text{Var} \to \text{Val} \to \text{Store} \to \text{Store}$
$\text{update} = \underline{\lambda} id.\underline{\lambda}\text{v}.\underline{\lambda}\sigma.[id \mapsto \text{v}]\sigma$

lookup: $\text{Var} \to \text{Store} \to \text{Val}$
$\text{lookup} = \underline{\lambda} id.\underline{\lambda}\sigma.\sigma\ id$

Valuation function:

$\mathbf{P} : \text{Prog} \to \text{Val} \to \text{Val}$
$\mathbf{P}\ [\![\text{Prog } c\ id]\!] = \underline{\lambda}\text{v}.\mathbf{C}[\![c]\!](\underline{\lambda} id.\text{v})[\![id]\!]$

$\mathbf{E} : \text{Exp} \to \text{Store} \to \text{Val}$
$\mathbf{E}[\![\text{Ident } id]\!] = \underline{\lambda}\sigma.\text{lookup } [\![id]\!]\ \sigma$
$\mathbf{E}[\![\text{Int } n]\!] = \underline{\lambda}\sigma.\mathbf{I}[\![n]\!]$
$\mathbf{E}[\![\text{Add } e\ e']\!] = \underline{\lambda}\sigma.(\mathbf{E}[\![e]\!]\sigma) + (\mathbf{E}[\![e']\!]\sigma)$
$\mathbf{E}[\![\text{Sub } e\ e']\!] = \underline{\lambda}\sigma.(\mathbf{E}[\![e]\!]\sigma) - (\mathbf{E}[\![e']\!]\sigma)$
$\mathbf{E}[\![\text{Mul } e\ e']\!] = \underline{\lambda}\sigma.(\mathbf{E}[\![e]\!]\sigma) * (\mathbf{E}[\![e']\!]\sigma)$

$\mathbf{C}$: $\text{Com} \to \text{Store} \to \text{Store}$
$\mathbf{C}[\![\text{Skip}]\!] = \underline{\lambda}\sigma.\sigma$
$\mathbf{C}[\![\text{Assign } id\ e]\!] = \underline{\lambda}\sigma.\text{update } [\![id]\!]\ (\mathbf{E}[\![e]\!]\sigma)\ \sigma$
$\mathbf{C}[\![\text{Seq } c_1\ c_2]\!] = \underline{\lambda}\sigma.\mathbf{C}[\![c_2]\!]\ (\mathbf{C}[\![c_1]\!]\sigma)$
$\mathbf{C}[\![\text{If } e\ c_1\ c_2]\!] = \underline{\lambda}\sigma.(\mathbf{E}[\![e]\!]\sigma{=}0) \to \mathbf{C}[\![c_1]\!]\sigma \square \mathbf{C}[\![c_2]\!]\sigma$
$\mathbf{C}[\![\text{While } e\ c]\!] =$
$\quad \mathbf{fix}\ (\lambda\text{wc}.\underline{\lambda}\sigma.(\mathbf{E}[\![e]\!]\sigma{=}0) \to \text{wc } \mathbf{C}[\![c]\!]\sigma \square \sigma)$

$\mathbf{I}$: $\text{Integer} \to \text{Val}$ (omitted)

$\mathbf{B}$: $\text{Val} \to \text{Val} \to \text{Val}$ (omitted)

Figure 19: Semantics of Tiny

	Meta(Def,P)	Int(P)	Target
Run times/ms:	133.0	5.14	0.15
Storage/bytes:	26916	12284	160

Figure 20: Performance of the Small target program

The run time of the target program is in fact quite good since the run time of an equivalent

handwritten Scheme program is 0.12 ms. compared to our 0.15 ms.

Runtime for the factorial program in Tiny can be seen in Figure 21. Again the input was 10 in all cases. The program used in the experiment was again the factorial program (here in concrete syntax):

```
program factorial(input,output);
begin
  x := input;
  fac := 1;
  while x
    fac := x*fac;
    x := x-1
  endwhile;
  output := fac
end.
```

	Meta(Def,P)	Int(P)	Target
Run times/ms:	106.3	3.7	0.49
Storage/bytes:	21144	7240	408

Figure 21: Performance of the Tiny target program

In general the speedup gained by specializing the meta-interpreter with respect to the definition is around 25. This seem to be of the same order of magnitude as is usually observed when specializing interpreters, though slightly on the large side. The speedups gained by specializing the interpreters, though still in the usual interval, differ a lot between the two languages. The speedup for the Small-interpreter is as high as 34, while it is "only" about 8 for the Tiny-interpreter. That the speedup is smaller for the Tiny-interpreter is not a surprise since the store operations are not remove by the compilation. That the speedup for the Small-interpreter is as high as it is, is in fact a surprising good result.

The importance of the experiments is that it seem to be possible to specialize meta-interpreters twice and get good results.

6.2.3 Generating the Compilers

The time used to generate the interpreter generator Intgen was 4.68 sec. and the size it is 2941 cells. (the number of "cons" cells needed to represent it in memory). Figure 22 shows the relevant times concerning the generation of the compilers and the sizes of the generated programs.

Output	Run	Run-times		Sizes	
		Small	Tiny	Small	Tiny
Int	= Intgen(Def)	230 ms.	160 ms.	477 cells	393 cells
Compiler	= Cogen(Int)	4.24 sec.	2.75 sec.	2775 cells	2015 cells
Target	= Compiler(fac)	16.3 ms.	18.8 ms.	54 cells	157 cells

Figure 22: Generation times

6.3 Meta-Interpreters Versus Interpreters

The strength of the meta-interpreter approach is that it is conceptually simple and that is should be reasonably easy to prove the correctness of meta-interpreters, at least compared to proving the correctness of a compiler generation directly. The correctness of the partial evaluator that is used will then ensure the correctness of the compiler generator. This is something that would probably be very difficult to do for most compiler generation systems. Another strength of the meta-interpreter approach is that we can use the meta-interpreter to restrict the way the interpreters may look. There are several reasons why it is is not a good idea to let users of a compiler generation system write languages definitions as interpreters in Scheme. Firstly, Scheme is not the clearest language to write in, denotational semantics is far better. Secondly, if Scheme is hard to write in for the language implementer, it is even harder to read for someone who only wants to become acquainted with the semantics of a language. Also here denotational semantics would be far better. Finally, we have no control over the the way interpreters may look. These may have very bad separation of binding times and result in poor compilers. Since the generated interpreters are specialized versions of the meta-interpreters we may ensure, by clever coding of the meta-interpreters that the interpreters get good separation of binding times. We may also add facilities to the meta-interpreters to guide the generation of interpreters and to report errors or possible inefficiencies in the definitions back to the users. A obvious possibility would be to type-check the definitions, but one could also let the user declare domains as being run-time (corresponds to having binding time dynamic in the interpreter) or compile-time (corresponds to having binding time static or closure in the interpreter) and let the system check this. In general we may say the partial evaluators are overly general for the purpose of compiler generation and meta-interpreters are a way to restrict this generality.

Small experiments have shown that it seems quite easy to change and extend the meta-interpreter to include new standard domains and operations on these, or to provide the user with the possibility to define both new domains and operations.

Denotational semantics is far from being the only definition language that the method describe is applicable to. One might expect that using a more specialized definition language would lead to even better results, since this may restrict the way the interpreters may look even further. Action semantics seems to be a likely candidate for a definition language in future experiments.

6.4 Conclusion

The results presented in this chapter, though preliminary, are very promising and show that there is hope for success with compiler generation systems based on partial evaluation of meta-interpreters.

Chapter 7

Binding Time Improvements

A common experience among people working with partial evaluation is that not all programs are equally well suited for partial evaluation and that a certain amount of rewriting is necessary to make a program "partially evaluate well." This means that much of what is presented in this chapter is in no way particular to partial evaluation of interpreters. The rewritings are semantics preserving transformations that aim to make more parts of a program static such that the partial evaluator can do more work, hence the name "binding time improvements". In this chapter we will describe some binding time improvements, that experience have shown us to be important, to improve target programs produced by partial evaluation generated compiler. These are:

- Using static information about dynamic data.
- Transformations into continuation passing style.
- Simulating partially static data structures to achieve *variable splitting* [Sestoft 1985].

Binding time improvements relate very much to the method of partial evaluation used. If a partial evaluator can treat partially static structures then there is no need for binding time improvements that achieve variable splitting (like the third item above).

A binding time improvement is correct if the programs before and after the improvement are semantically equivalent, i.e. computes the same function. What we hope to gain from the improvement is better residual code. Usually this means that the target programs produced by our compilers after the improvement runs faster, uses less memory and/or are smaller. A binding time improvement of a program may in fact not be an improvements of the program. It may in fact reduce the performance of both the programs and of in case of interpreters also the performance of the compilers.

We also have to say something about termination of partial evaluation. By making more things static we risk nontermination of the partial evaluation process, so we have to show that termination of partial evaluation is preserved by the improvement in order to prove correctness. We say that we have partial correctness when we cannot guarantee termination of partial evaluation.

We will now explain the improvements in detail.

7.1 Using Static Information about Dynamic Data

A key point in this rewriting process is to identify static information that is not visible to the specializer, and make this visible. Let us assume for instance that we have some dynamic variable `x` somewhere in our program. Since `x` is dynamic the specializer cannot do any computation

depending on `x`. Now it might be possible from the context in which `x` occurs to detect some information about the value of `x` depending only on static values in the program and then rewrite the program such that the specializer can utilize this information. This was one of the main ones steps in the KMP derivation by Consel and Danvy [Consel and Danvy 1989]. The simplest example is a conditional with an `equal?` test:

```
(if (equal? s d)
    (f d)
    ...)
```

If `s` is static and `d` is dynamic we may improve the binding times of the enclosing program by replacing `d` by `s` in the "then" branch of the conditional. The example might seem silly, since no one will ever write the expression like that in the first place, but as we shall see later it may actually be a key point in many more complex improvements.

Let us now look at a more realistic example. The example shows a rewriting that is a classic (it was used already in the original MIX project [Jones *et al.* 1985]). Because of this and its widespread use, we have coined the name "The Trick" for this optimization. Assume we have an expression:

```
(if (member x l) (f x) (g x))
```

somewhere in a program that we want to specialize. Assume further that `x` is dynamic and `l` static. Then the formal parameter of `f` will also be dynamic. But we know that `x` can only be one of the values in `l` and the value of `l` is known at partial evaluation time. If we rewrite the expression to:

```
(member1 x l)
```

and introduce a new function `member1` defined by:

```
(define (member1 x l)
  (if (null? l)
      (g x)
      (let ((l1 (car l)))
        (if (equal? l1 x)
            (f l1)
            (member1 x (cdr l))))))
```

we will get an equivalent program, but now `f` is called with a static argument and `f`'s formal parameter is now not required to be dynamic by this part of the program (there might be other calls to `f` that make the parameter dynamic and clearly one should only use the transformation if it actually makes `f`'s formal parameter static).

The idea of using static information about dynamic data is used extensively in [Jørgensen 1991] where a pattern matching interpreter is rewritten to generate a compiler that produces efficient matching code.

7.1.1 Folding Conditionals over Predicates

We will now try to make the optimization above less ad. hoc. and see if we can automate it. To be able to automate the optimization we must at least know the structure of `member`, that is `member` has to be part of the program that is being specialized, not a primitive operation. Lets assume that we know `member` and its defined as follows (notice that this definition is not equivalent to the standard Scheme `member` function):

```
(define (member e l)
  (if (null? l)
      '#f
      (if (equal? e (car l))
          '#t
          (member e (cdr l)))))
```

and assume that the code that we want to transform has the structure:

```
(if (member x l) e1 e2)
```

We define a new function `member1` and replace the code by:

```
(member1 x l (lambda (x) e1) (lambda (x) e2))
```

and define `member1` as:

```
(define (member1 e l c1 c2)
  (if (null? l)
      (c2 e)
      (let ((e1 (car l)))
        (if (equal? e e1)
            (c1 e1)
            (member1 e (cdr l) c1 c2)))))
```

The questions are now: how to obtain functions like `member1` from `member`, when this is possible and when this will result in a binding time improvement? We will treat the first question first. Figure 23 shows a transformation T that when applied to `member` above gives `member1`. The transformation should only be applied when a certain on the body of the predicate condition holds. This condition is in fact just that it should be well formed. Since Scheme is dynamically typed there is no guarantee of this, so we have to check this before applying the transformation. In this context we also assume that that the normal logical relations `and`, `or`, etc. have been expanded into equivalent code using conditionals. We could have relax a little on the condition by using the usual Scheme convention that any value different from false represent true, but this is not important for our treatment here. A function that checks if this condition hold is defined in Figure 24. This then tells us when the translation is possible, but not when it is profitable. This may in general depend on large parts of the program in which the expression is contained and there seems to be no other solution than to redo the binding time analysis. An other problem is that the transformation might be applicable even in cases where non of the arguments to the test are static. This can happen when one of the arguments become static as an effect of the transformation. This is in fact the case for the regular interpreter that we have developed in collaboration with Anders Bondorf and Torben Mogensen (both from DIKU). We will not give a description of the example (one can be found in [Bondorf 1990b]). In this example the predicate was also `member`, and both the element and the list were dynamic before the transformation. However, after the transformation the list became static and the optimization proved fruitful. This shows that it can be very hard to reason about when this improvement should be applied or not.

A simple way to use this transformation is the following. Assume that we have an expression e depending on a dynamic variable `x` in a program. If we can define a function `f` that from static values in the program can calculate all the possible values that `x` can have in e, then we can replace e by:

Consider an expression:

`(if (p x` e`)` e_c e_a`)`

where `x` is dynamic, e will become static after the transformation, `p` is defined by:

`(define (p x y)` e_p`)`

and the condition $\mathrm{C}[\![e_p]\!]$ holds. This is transformed into:

`(p1 x` e `(lambda (x)` e_c`) (lambda (x)` e_a`))`
`(define (p1 x y c1 c2)` $\mathtt{T}[\![e_p]\!]\emptyset$`)`

where

$\mathrm{T}[\![$ `'#t` $]\!]\Gamma =$ `c1 x1`, **if** $[$`x1/x`$] \in \Gamma$
`c1 x`, **otherwise**
$\mathrm{T}[\![$ `'#f` $]\!]\Gamma =$ `c2 x`
$\mathrm{T}[\![$ `(equal? x` e`)` $]\!]\Gamma =$ `(let ((x1` e`)) (if (equal? x x1) (c1 x1) (c2 x)))`
$\mathrm{T}[\![$ `(if (equal? x` e_1`)` e_2 e_3`)` $]\!]\Gamma =$ `(let ((x1` e_1`))`
`(if (equal? x x1)` $(\mathrm{T}[\![e_2]\!]\{$`[x1/x]`$\}\cup\Gamma)$ $(\mathrm{T}[\![e_3]\!]\Gamma)))$
$\mathrm{T}[\![$ `(if` e_1 e_2 e_3`)` $]\!]\Gamma =$ `(if` e_1 $(\mathrm{T}[\![e_2]\!]\Gamma)$ $(\mathrm{T}[\![e_3]\!]\Gamma)$`)`
$\mathrm{T}[\![$ `(p x` e`)` $]\!]\Gamma =$ `(p1 x` e `c1 c2)`
$\mathrm{T}[\![$ `(f x` e`)` $]\!]\Gamma =$ `(f1 x` e `c1 c2)`
where `f` is defined by: `(define (f x y)` e_f`)`
and `f1` is a new function defined by:
`(define (f1 x y c1 c2)` $\mathtt{T}[\![e_f]\!]\emptyset$`)`

Figure 23: Binding time transformation

$\mathrm{C}[\![$ `'#t` $]\!] =$ True
$\mathrm{C}[\![$ `'#f` $]\!] =$ True
$\mathrm{C}[\![$ `(equal? x` e`)` $]\!] =$ True
$\mathrm{C}[\![$ `(if` e_1 e_2 e_3`)` $]\!] = (\mathrm{C}[\![e_2]\!]) \wedge (\mathrm{C}[\![e_3]\!])$
$\mathrm{C}[\![$ `(p x` e`)` $]\!] =$ True
$\mathrm{C}[\![$ `(f x` e`)` $]\!] = \mathrm{C}[\![e_f]\!]$
where `f` is define as: `(define (f x y)` e_f`)`
$\mathrm{C}[\![e]\!] =$ False, **otherwise**

Figure 24: Check for applicability of transformation

```
(let ([l (f <static arguments>)])
  (if (member x l) e ()))
```

and let the transformation do the rest of the work. The result of this will be a residual code of the form:

```
(cond
 ((equal? x l1) re1)
 ((equal? x l2) re2)
 ...)
```

where re_i is a specialized version of e with `x` replaced with `li`. In a way it corresponds to a tabulation of e for all the possible values of `x`.

7.1.2 Correctness

The transformation can be viewed as composed by two separate translations. The first one transform the conditional and the function `p` into (a restricted form of) continuation passing style. The second one transforms simple tests like (as in the simple example above):

```
(if (equal? x e) e1 e2)
```

where `x` is dynamic and e static, into:

```
(let ([x1 e])
  (if (equal? x x1) e1[x1/x] e2))
```

The correctness of the transformation as a whole then depends on the correctness of these two transformations. The last one is trivial while the first one depends on whether the transformation into continuation passing style is correct.

7.2 Continuation Passing Style

One of the most important rewritings is transformation into continuation passing style (in some cases just tail recursive form). The importance of this has also been shown in [Danvy 1991] and [Jørgensen 1990]. The cases that can often successfully be handled by this kind of transformation are those where static values get caught in a dynamic context (a specialization point). A typical example is a function returning several results (packed together in e.g. a list) of which some are static and some dynamic. The result will be a dynamic value and the static values will then be inaccessible to the specializer. This is not the case with a specializer that treats partially static structures. By a transformation into continuation passing style one can pass the results separately to a continuation which can now use the static ones of these.

Another situation in which transformation into continuation passing style may improve binding times is when functions that represent specialization points (i.e. become residual functions) return higher order values. Let us look at an example (a trivial one, serving only to demonstrate the idea). Consider an application

```
((f x) 42)
```

where `f` is defined by:

```
(define (f x)
  (if < some test involving x >
      (lambda (z) (sub1 z))
      (lambda (z) (add1 z))))
```

Assume that `x` is dynamic, then calls to `f` will be specialization points (with Similix's current strategy) and Similix will therefore make `(f x)` dynamic. Hence the application of `(f x)` to `42`

will not be performed. On the other hand, if we rewrite the application into:

```
(fc x (lambda (c) (c 42)))
```

and the definition of `f` into:

```
(define (fc x c)
  (if < some test involving x >
      (c (lambda (z) (sub1 z)))
      (c (lambda (z) (add1 z)))))
```

Then application of the lambda abstractions to `42` will be performed, since the continuation is applied directly to the closure values that the original function `f` would have returned.

Methods to make this transformation automatic are described in [Consel and Danvy 1991a] and [Holst and Gomard 1991].

7.2.1 Correctness

Assume that we are given a partial evaluator that obeys the MIX-equation. Since CPS-conversion preserves the semantics we can assume correctness of the transformation for that particular partial evaluator. But most partial evaluators do not obey the MIX-equation due to possible nontermination of the partial evaluation process, so in that case we can only prove partial correctness. An example of how we can get nontermination of partial evaluation by transforming into continuation passing style is the `append` function. If we rewrite `append` into continuation passing style we get:

```
(define (append x y)
  (appendc x y (lambda (l) l)))

(define (appendc x y c)
  (if (null? x)
      (c y)
      (appendc (cdr x) y (lambda (l) (c (cons (car x) l))))))
```

If we specialize this version of append with respect a dynamic first argument and a static second argument, then we will get infinite specialization. The reason is that the continuation gets bigger and bigger for every specialized version of `appendc` that Similix generates. Similix is in other words trying to generate specialized versions of `append` for all possible lengths of the list input `x`. This means that every time we use transformation into continuation passing style as a binding time improvement we have to make sure that partial evaluation terminates.

7.3 Simulating Partially Static Data Structures

As stated in Chapter 1, Similix does not handle partially static data structures, but in many cases these can be simulated by using higher order functions. The structures presented in this section all have dynamic structure, but the binding time values of the elements in the structures are all the same (dynamic or static). This means that we cannot handle for instance pairs of a dynamic and a static component, but as we have already seen these can be handled by a transformation into continuation passing style.

The simplest example is partially static lists which can be simulated by using the definitions shown in Figure 25 (this special version is due to Torben Mogensen) instead of `()`, `cons` and `list-ref`.

```
(define (snil) (lambda (n c) (n)))

(define (scons a d) (lambda (n c) (c a d)))

(define (slist-ref l index)
  (l
   (lambda () 'error)
   (lambda (a d)
     (if (= 0 index)
         a
         (slist-ref d (sub1 index))))))
```

Figure 25: Simulating partially static lists

The effect gained by this is normally called variable splitting or *arity raising* [Romanenko 1988]. The reason that this method works is that Similix treats higher order functions, called closures, as a kind of partially static structures, that is, they may contain both static and dynamic components. When a closure is applied to its arguments during specialization, the body of the closure is specialized and the substructures of the body may emerge again.

An other way to do this improvement is to replace structures by environments. The way structures can be represented in general is by having the environment map an occurrence to the value at the occurrence. For lists this means that one can use the positions (indexes) in the lists as occurrences. If `rl` is an environment replacing a list `l` then the reference to an element in the list `(list-ref l n)` should be replaced by `(rl n)`.

Let us look at an example: if we want to represent a binary tree by an environment we could do this by using occurrences defined by the following syntax:

tree → `leaf` *info* | `node` *info tree tree*
tree-occ → `info` | `left` *tree-occ* | `right` *tree-occ*

This means that if the tree `(node 1 (leaf 2) (leaf 3))` is represented by the environment ρ, then we have ρ(`info`) = `1`, ρ(`left`) = `(Leaf 2)`, ρ(`right`) = `(leaf 3)`, ρ(`left info`) = `2` and ρ(`right info`) = `3`.

Partial evaluators that handle partially static structures will of course not need any of these binding time improvements.

7.3.1 Correctness

We will show under what conditions the improvement in Figure 25 is correct. One can prove the following:

`(slist-ref (scons` $expr_1$ $expr_2$`) 0)` = $expr_1$
`(slist-ref (scons` $expr_1$ $expr_2$`)` n`)` = `(slist-ref` $expr_2$ `(sub1` n`))`, **if** $n > 0$
`(slist-ref snil` n`)` = `'error`

This shows that the new definitions behave like `()`, `cons` and `list-ref` as long as one does not use an index that is out of range (Scheme would give an error message when evaluating `(list-ref () 0)`). The reason that we use `'error` instead of a call to the Scheme function `error` is that the former is static while the latter is dynamic. Using a call to `error` would make the binding time of all the elements in lists dynamic.

This means that we have correctness of the improvement when we can make sure not to get an index out of range error. Using similar arguments one can show the correctness of the binding time improvement using environments.

Chapter 8

Optimizations in Compilers

Ideally, compilers should produce target code that is as good as we would write it by hand, but this is often far from the case, especially, when compilers are generated using partial evaluation. In usual compiler technology one tries to improve the quality of the target code by adding program transformations that are traditionally called *optimization*. These optimization try to make programs run faster, use less space, be smaller, etc. Usually the a optimization phase is added to compilers, but what we will look at in this chapter is how optimizations can be used in connection with compilers generated by partial evaluation; what kind optimizations that can be used; and if necessary, what kind of analyses that are needed. We will show how we can make our generated compiler do typical compiler optimizations by changing our interpreters in suitable ways. The main idea is that we make the interpreter collect and use information during interpretation. The collect information is used to do simple optimization during interpretation. If the collection and use of information can be made completely static, we can make in take place at compile time (specialization time), and the generated compiler will produce optimized code. In this way we do not get a separate optimization phase.

We divide the optimizations into three groups on account of their need for contextual information. We call an optimizations *on-line* if it depends only on information collected during evaluation. By this we mean, when we evaluate an expression we traverse the expression in some way to evaluate its subexpressions. During this we may collect information about the subexpressions that we may use when evaluating other subexpressions or the expressions itself. We call an optimizations *local* if it depends on an analysis of only the syntax phrase being interpreted. Otherwise, if the optimization depends on information that can only be obtained by a global analysis, we call the optimization *global*. The reason that we want to make this distinction is that on-line optimization and partly local optimization are the ones that we can perform completely during interpretation of a program, while global optimization depends on a global analysis that we might as well do before the interpretation of a program.

Analyses will in general slow down the interpretation process, since it will often take longer to collect information than the time saved by using the information. Even worse is the fact that the analysis of an expressions is performed every time the expression is evaluated. This is however not a problem when using partial evaluation to compile. First of all, the analyses are performed during compilation, since they depend only on static data and is therefore not a burden at runtime. But more importantly, they are only performed once for each expression, since the interpretation of a given expression is only specialized once. If we did global analysis during interpretation, we will have to do the same work many times even when using partial evaluation.

In the following we will discuss how optimizations in the different groups can be combined with the process of doing compiler generation by partial evaluation. We will also discuss what kind of optimizations fall in the three groups.

8.1 On-line Optimizations

On-line optimizations is done by collecting (during evaluation) information about the values of variable or expressions. Examples could be that a variable has a certain constant value or an expression has a certain value in a certain scope etc.

Constant folding is an example of transformation that can be done by an on-line optimization. During evaluation we can collect information about which expressions have a constant value and what their values are. Then an expression can be evaluated completely at compile time if all its subexpressions have constant values. This of course introduces the usual problem of nontermination, but this is not different from constant folding in conventional compiler techniques and can be solved by restricting what kind of expressions are going to be evaluated.

An optimization of similar nature is common subexpression elimination. We keep track of the value of all compound expressions (not constant or variable) during evaluation. We also keep track of the "scopes" in which the expressions had their values. Every time we evaluate a new compound expression we check if we have encountered it before and if the scope was the same as the current one. If so we reuse the stored value. Figure 26 shows a modified version of the interpreter from section 5.7. The interpreter is now for a call by value language and does *memoization* of the values of compound expressions. The problem with "scopes" are handled by throwing away all information whenever new bindings are introduced. The variable `ce` is the set of compound expressions seen so far and `re` is the environment mapping expressions to values. The function `fix` is unchanged and is therefore left out.

If we generate a compiler from the interpreter and compile the program:

```
(* (+ 1 y) (* 2 (+ 1 y)))
```

we get the following target program:

```
(define (run-0 w_0)
  (let ([v3_1 (+ 1 w_0)]) (* v3_1 (* 2 v3_1)))))
```

In order to obtain this result it is necessary to translate evaluation of expression into continuation passing style as it can be seen in Figure 26. This is because the evaluation of an expression now gives as result a triple consisting of the usual result, the set of compound expressions evaluated, and an environment mapping compound expressions to values. As explained in Chapter 7 this is one of the cases where translation into continuation passing style is needed to separate static data (the set of expression and the environment) from dynamic data (the result). The let-expression inserted in the target program comes from the interpreter and is not unfolded by Similix, since Similix is careful not to duplicate evaluation. If the value is only used once, Similix will unfold the let-expression. In this way we are utilizing the occurrence count analysis that Similix performs, without having to bother about it in details.

A final example of an on-line optimization is insertions of force operations (operations that force evaluation in a lazy language). This corresponds somehow to a local evaluation order analysis. We will described this in details in Chapter 14.

8.2 Local Optimizations

Local optimizations may use any information collected during evaluation (like on-line analyses), but can additionally look ahead on the syntax object being interpreted. This could be to find the free variables in a block of code or to find the variables in which an expression is strict. What a local optimizations cannot do is to take global properties into account, that is, properties of any part of a program other than the syntax object being interpreted.

```
(define (run e w) (_E e (lambda (v) w) '() (init-ceenv) (idc)))
;
; _E: Exp -> Env -> List(Exp) -> CEnv -> Cont -> Val
(define (_E e r ce re c)
  (cond
   ((member e ce) (c (re e) ce re))
   ((EVar? e) (c (r (EVar->Var e)) ce re))
   ((EConst? e) (c (_C (EConst->Const e)) ce re))
   ((EBinop? e) (_E (EBinop->E1 e) r ce re
                    (lambda (v1 ce1 re1)
                      (_E (EBinop->E2 e) r ce1 re1
                          (lambda (v2 ce2 re2)
                            (let ([v3 ((_B (EBinop->Binop e)) v1 v2)])
                              (c v3 (cons e ce2) (upd e v3 re2)))))))))
   ((EIf? e) (_E (EIf->E1 e) r ce re
                 (lambda (v ce1 re1)
                   (if v
                       (_E (EIf->E2 e) r (cons e ce1) (upd e v re1) c)
                       (_E (EIf->E3 e) r (cons e ce1) (upd e v re1) c)))))
   ((ELambda? e) (c (lambda (w)
                      (_E (ELambda->E e)
                          (upd (ELambda->Var e) w r)
                          '() (init-ceenv) (idc)))
                    ce re))
   ((ELetrec? e) (c (_E (ELetrec->E2 e)
                        (fix (lambda (r1)
                               (upd (ELetrec->Var e)
                                    (_E (ELetrec->E1 e) r1
                                        '() (init-ceenv) (idc))
                                    r)))
                        '() (init-ceenv) (idc))
                    ce re))
   (else         (_E (EApply->E1 e) r ce re
                     (lambda (f ce1 re1)
                       (_E (EApply->E2 e) r ce1 re1
                           (lambda (v ce2 re2)
                             (let ([v1 (f v)])
                               (c v1 (cons e ce2) (upd e v1 re2)))))))))
(define (idc) (lambda (v ce re) v))
(define (init-ceenv)
  (lambda (w) (error 'init-ceenv "~s" w)))
(define (upd v w r)
  (lambda (v1) (if (equal? v v1) w (r v1))))
```

Figure 26: Interpreter with common subexpression optimization

The simplest kinds of local optimizations are *peephole optimizations*. This might be special treatment of some syntactical cases.

8.2.1 Special syntactical cases

Sometimes there may be syntactic cases that we want to treat specially efficient. In abstract syntax one often seek uniformity in phrases, for example in a language with guarded alternatives (like BAWL, see Chapter 10) all alternatives will have a guard, even though this is not the case

in the concrete syntax. An example could be the following definition:

```
succ x = x+1
```

The abstract syntax for this corresponds to inserting a guard `True` and even an empty list of local definitions. This can be illustrated informally like:

```
succ x = x+1, True
         where {}
```

If we were to interpret this directly in our interpreter, then the target program generated by the corresponding compiler would look like[1]:

```
(define (succ)
  (lambda (v_5)
    (let ([cfst_9 (make-cfst 0)]) ; code related to the local definitions
      (if '#t                     ; conditional corresponding to the guard
          (((B '!plus) (lambda () (v_5)))
           (lambda () 1))
          (error ...)))))
```

Here `B` is a primitive operation interpreting basic operators. The variable `cfst_9` corresponds to the local definitions and the test `#t` to the guard. It is clear that this target function is an unsatisfactory result. If the interpreter had a test:

```
(if (isTrue? guard)
    (E expr ...)
    (if (E guard ...)
        (E expr ...)
        ...))
```

then the test `isTrue? guard` which test whether the guard expression is `True` will be static and we will not get the test in the target program. A similar optimizations of the interpreter can be applied to the case of empty local definitions. The target program would the look something like:

```
(define (succ)
  (lambda (v_5)
    (((B '!plus) (lambda () (v_5))) (lambda () 1))))
```

which is much better. The example also shows another possible local optimization. Instead of "interpreting" operators like `+`, `-`, etc., at runtime we would like to have the operator inlined. That is we would like to have target code that looked like:

```
(define (succ)
  (lambda (v_5)
    (+ (v_5) 1)))
```

This is in fact possible and the final BAWL compiler presented in the second part of this thesis uses this optimization. On in fact just have to do two things. First, make `B` a part of the program that the partial evaluator see and not a primitive operation as it have been until now. Second, interpret complete application of operators in a special way.

[1]The function `make-cfst` and other details in this code are not important for our discussion here; they will be described in the second part of this thesis.

Some more examples of this will be given in Chapter 14.

8.2.2 Restriction of environments

A less trivial example of a local optimization is based on a simple kind of live variable analysis. For a functional language this corresponds to finding the variables that an expression may depend on, for example the free variables of the expression. This optimization may be used to reduce the size of of environments during interpretation, which in turn may reduce the arity of the produced target programs.

We will now explain this in details. Assume that our source language is a functional language like BAWL and that we have a definition of the form:

```
f x y = ... (fac x) ...
        where
          fac 0 = 1
          fac n = n*fac(n-1)
```

in a source program. Then a compiler generated from a simple interpreter might give a result of form:

```
(define (fac-0 x_0 y_0)
  (lambda (n_0)
    (...
        (fac-0 x_0 y_0))))
```

The variables `x_0` and `y_0` reflect that the values of `x` and `y` are in fact accessible inside the definition of `fac`. The specializer is "unaware" of the fact that the definition of `fac` does not use these values and therefore "thinks" that a call to `fac-0` may depend on these. This problem is also mentioned by Bondorf in [Bondorf 1990a], but no solution was given there.

A way to solve the problem is to do a live variable analysis of `fac`'s definition. The result of this analysis is then used to reduce the set of variables that are bound in environments to the set of live variables. This reduction then has to take place before evaluating `fac`'s right hand side. This will be described in detail in Section 14.2.2.

8.2.3 Local insertions of force operations

One would expect strictness analysis to be a global analysis and it usually is, but if we are ready to accept a less precise but safe result then we can make it local. This may seem a little strange since the usual use of strictness analysis results is clearly non local. However, we can use strictness analysis in conjunction with a usage count analysis to get a local evaluation order analysis, and the kind of strictness information we are interested in is what variables an expression is strict in. We can use this analysis to insert points in an expression where a given variable can safely be forced and saved for later use. Such a point corresponds to a subexpression which is strict in the variable and the variable may be used more than once during some evaluation. An example could be:

```
if (x=y)
   0
   (x+z)
```

In this expression, both `x` and `y` may be forced before evaluating the expression, but only `x` is may be used more than once during some evaluation. We will not go further into details with the analysis, but as it may be apparent from the discussion this is not such a simple analysis.

As mentioned in Section 8.1 we may achieve the same result using an on-line optimization and this turns out to be much simpler to implement (see Chapter 14).

8.3 Global Optimizations

Global optimizations are those that need more than just information that can be found by on-line and local analyses. This means that there is a need for some global analysis and we might as well do this before the interpretation of a program, since we do not want to redo the whole analysis every time we need some piece information.

It might of course be possible to do global analyses "by need" when interpreting. That is, every time we need some information that an analysis should supply us with, we do as much of the analysis as is needed to obtain the information. The part of the analysis that had to be performed to do so should then be saved and reused if the information is ever needed again. This corresponds to finding a partial fixed point for the analysis. Partial evaluation of a program with such an analysis should then result in finding the complete fixed point (or as much as will ever be needed). However, there is a problem with *dead static data*, static data that the has no influence on the residual code, when using this method. If some code is analyzed twice or more, then the first time there is no information available from the analysis, but the following times there are. This means that the static data is different the first time and the following times and the code generated will not be shared as it should be.

Many of the usual analyses used in compiler construction are global analyses. Clearly analyses that involve finding a fixed point of some functional depending on an entire program belongs to this class. Examples of such analyses are strictness analysis, update analysis and uncurrying analysis. We will not go into details of how to do global analyses, but we will show how the result of an analysis can be used by in an interpreter to yield a compiler with a global optimization. We will look at strictness analysis in the next section.

8.3.1 Adding strictness optimization to an interpreter

Strictness analysis tells us when we can safely evaluate an argument before an application. Let us return to the small example language from Section 5.7. The factorial function may be computed as follows in this language.

```
(letrec fac
        (lambda x
          (if (= x 0)
              1
              (* x (fac (- x 1)))))
        (fac v))
```

If we generate a compiler from the interpreter of Section 5.7 and compile the factorial program we get the following target program:

```
(define (run-0 w_0)
  (((_e-1-2 w_0)) (lambda () (let ([w_2 (_e-1-2 w_0)]) w_0))))
(define (_e-1-2 r1_0)
  (lambda ()
    (lambda (w_1)
      (if (= (w_1) 0)
          1
          (* (w_1) (((_e-1-2 r1_0)) (lambda () (- (w_1) 1))))))))
```

This program can be optimized since the factorial function is strict. We introduce the annotation `x!` to mean that the value of `x` is always evaluated in the environment. We call such variables "strict" variables. We also introduce the annotation (e_1 `@!` e_2) to mean strict application. When the interpreter interprets a strict application it evaluates the argument before doing the application. Here is an annotated version of the factorial program:

```
(letrec fac!
        (lambda x!
          (if (= x! 0)
              1
              (* x! (fac! @! (- x! 1)))))
        (fac! @! v))
```

Figure 27 shows a modified version of the interpreter of Section 5.7 that does strictness optimization.

There are three modifications that have to do with the strictness optimization. Firstly, interpretation of variables has been changed to:

```
((EVar? e)    (if (strictVar? e)
                  (r (EVar->Var e))
                  (my-force (r (EVar->Var e)))))
```

That is, a strict variable, tested by `strictVar?`, is not forced when looked up. Secondly, interpretation of application has been changed to:

```
((EApply? e)  ((_E (EApply->E1 e) r)
               (if (strictApply? e)
                   (_E (EApply->E2 e) r)
                   (my-delay (_E (EApply->E2 e) r)))))
```

so that arguments to strict application are not suspended. Finally, the interpretation of letrec-expressions also has to be changed, since the letrec-expressions also introduces variables, that can be either strict or non-strict.

```
((ELetrec? e) (_E (ELetrec->E2 e)
                  (fix (lambda (r1)
                         (upd (ELetrec->Var e)
                              (if (strictLetrec? e)
                                  (_E (ELetrec->E1 e) r1)
                                  (my-delay (_E (ELetrec->E1 e) r1)))
                              r)))))
```

Generating a compiler from this new interpreter we get a compiler that does strictness optimization. The target program for the factorial program then looks:

```
(define (run-0 w_0)
  (let ([w_2 (_e-2-1 w_0)]) ((_e-2-1 w_0) w_0)))
(define (_e-2-1 r_0)
  (lambda (w_1)
    (if (= w_1 0)
        1
        (* w_1 ((_e-2-1 r_0) (- w_1 1))))))
```

```
(define (run e w)
  (_E e (lambda (v) (my-delay w))))
;
; _E: Exp -> Env -> Val
(define (_E e r)
  (cond
   ((EVar? e)    (if (strictVar? e)
                     (r (EVar->Var e))
                     (my-force (r (EVar->Var e)))))
   ((EConst? e)  (_C (EConst->Const e)))
   ((EBinop? e)  ((_B (EBinop->Binop e))
                  (_E (EBinop->E1 e) r)
                  (_E (EBinop->E2 e) r)))
   ((EIf? e)     (if (_E (EIf->E1 e) r)
                     (_E (EIf->E2 e) r)
                     (_E (EIf->E3 e) r)))
   ((ELambda? e) (lambda (w)
                   (_E (ELambda->E e)
                       (upd (ELambda->Var e) w r))))
   ((ELetrec? e) (_E (ELetrec->E2 e)
                     (fix (lambda (r1)
                            (upd (ELetrec->Var e)
                                 (if (strictLetrec? e)
                                     (_E (ELetrec->E1 e) r1)
                                     (my-delay (_E (ELetrec->E1 e) r1)))
                                 r)))))
   (else         ((_E (EApply->E1 e) r)
                  (if (strictApply? e)
                      (_E (EApply->E2 e) r)
                      (my-delay (_E (EApply->E2 e) r)))))))

(define (upd v w r)
  (lambda (v1)
    (if (equal? v v1)
        w
        (r v1))))
```

Figure 27: Interpreter with strictness optimization

when annotated as shown above. In this target program the function `(_e-2-1 r_0)` corresponds to `fac`. We can see that the target program looks as we would expect to look: the argument to `(_e-2-1 r_0)` is not being delayed and the parameter `w_1` of `(_e-2-1 r_0)` is not being forced.

Chapter 9

Discussion

This chapter concludes the first part of this thesis. We will do this by discussing the advantages and disadvantages of partial evaluation as a compiler generation tool in Section 9.1. We will discuss related works in Section 9.2, future work in Section 9.3 and finally conclude in Section 9.4.

9.1 Partial Evaluation as a Compiler Generation Tool

For a compiler generation system to be of any use it should either provide us with a faster way to obtain compilers than traditional methods or it should give us a safer method to obtain correct compilers.

If writing a specification is just as hard and error-prone as writing the actual compiler by hand then nothing is gained.

By the correctness of a compiler we will mean correctness with respect to the the original semantics. This correctness depends on the correctness of Similix; on the correctness of the binding time improvements and the other transformations of the interpreter; and the correctness of the transformation of the semantics into the interpreter.

Some of the advantages of using partial evaluation as a compiler generation tool are:

- It is generally easier to read and write an interpreter than a compiler. This makes it easier to change and maintain the compiler.
- Debugging an interpreter is easier than debugging a compiler. One can trace evaluation and one can look at both compile time data structures and run time data structures at the same time. When debugging a compiler either the compiler or the target program may fail to work; there is not this separation when debugging an interpreter. We say that there is no separation of binding times.
- One has an interpreter for free. That is, we get both an interpreter and a compiler, which is not the case if we just write the compiler directly. This can be useful for many purposes, especially one might instrument the interpreter with operations tracing the execution or doing statistics. This will the result in a compiler inserting these operations in the target programs.
- Code generation is handled by the partial evaluator, i.e. one does not have to think about things like label or variable name generation, back-patching etc.
- Target and specification language is the same (in our case Scheme). One less language to think about makes life easier and one may assume that the specification language is well known to the user of partial evaluation.

Some of the disadvantages are:

- One has less control over the code generation since it is done by the partial evaluator. This means that one can only generate code that the partial evaluator is able to produce. Also, one may not be able to obtain the sharing of target code that is possible by conventional compilers.

- Target and specification languages are the same. This is also a disadvantage since it prevents us from choosing different target languages.

In general one can say that we get the advantages at the price of some freedom in the way we can generate target code.

9.2 Related Works

Related work in compiler generation has been discussed in Section 3.1 an we will not discuss this further here, but only look at related work in compiler generation by partial evaluation. Experiments in this direction go back to the early days of partial evaluation, but completely automatic generation of compilers was first achieved by Jones et. al. [Jones *et al.* 1985] in 1985. Since then experiments in different directions have been made. Bondorf abd Danvy [Bondorf and Danvy 1991] generated a compiler for a imperative language using Similix. The target code operates directly on the Scheme store. It is exploited that Similix can handle side effects on global variables. Bondorf [Bondorf 1991b] has also made experiments with compiling a small lazy combinator language using Similix using the same method as we do for making application lazy. Consel and Danvy [Consel and Danvy 1991b] have generated a compiler for a small algol subset using the Schism partial evaluator. The main point was that their compiler did static type checking. In relation to our work presented in the second part of this thesis, none of the experiments above treated a language that was close to having a realistic size.

A few experiments with partial evaluation of meta-interpreters have be conducted. Experiments in this direction were carried out by Mogensen (then and now at DIKU) with an early version of the Mixwell partial evaluator [Jones *et al.* 1985], but since the Mixwell language was first order the results were not very satisfactory and involved a lot of clever tricks. Recently Dybkjær [Dybkjær 1991] has applied Similix to a meta-interpreter. His meta-language is based on category theory and the interpreters that he generated did not always have "good" binding time properties.

Binding time improvements have been used as long as partial evaluators have existed, but it only most recently that the subject has been studied in details. An exception is possibly Dybkjær [Dybkjær 1985] who discusses some of the binding time improvements that he needed to generate parsers by applying the Mixwell partial evaluator to a general parsing algorithm. Recent work on the subject may be found in: [Consel and Danvy 1989] [Holst and Hughes 1991] [Jørgensen 1991] [Consel and Danvy 1991a] [Holst and Gomard 1991] [Bondorf 1991c].

9.3 Future Work

A lot of interesting work remains to be done.

It remains to be shown whether it is possible to automate some or all of the binding time improvements discussed in Section 7. Binding time improvement is certainly an interesting field and presently some research is under way in this field. A interesting project would be to implement the translation of Chapter 5 and build a compiler generation system on top of this and Similix. The way to do this could be by using a meta-interpreter.

Another interesting project is of course to write a meta interpreter for some definition language, maybe some variant of action semantics and try to build the compiler generation system around this. The experiments reported in Chapter 6 certainly seem to predict that such a project will result in a good and realistic system.

Finally it would be interesting to develop and implement the type inference based translation of section 5.4.

9.4 Conclusion

Let us re-examine Figure 1 of Chapter 3 and let us look at the operations in the right side of this leading to the generation of the code generator. The experiments in Chapter 6 seems to suggest it should be possible to automate the translation of denotational definitions into interpreters with "good" binding time properties. This means that we may justifiably hope to automate the first of these operations.

The second operation is the binding time improvements. The newest version of Similix includes partially static structures which do not only remove the need for the improvements concerning these, but also some cases where we used transformation into continuation passing style. The latter cases are those where we used continuation passing style to pass on multiple results. This leaves us with one case of binding time improvements involving continuation passing style that we cannot currently automate (passing a static value out through a dynamic conditional. The binding time improvements that use static information about dynamic data may be harder to automate, but some research in this direction has been done, for example an idea called *generalized partial computation* [Futamura and Nogi 1988].

The last of the manual operations of Figure 1 is the one called operations. These may be hard to optimize, but some of these may however be done by general postprocessing, e.g. superfluous parameters.

So all in all we can conclude that it should be possible to automate a great deal of the manual operations described in this thesis and that it should be possible to build a relatively efficient compiler generation system using the ideas described.

Part II

A case study

Chapter 10

The BAWL language

We call our language BAWL (Bird and Wadler Language), since it is modeled after the language in the book by Bird and Wadler [Bird and Wadler 1988]. The closest relatives are KRC [Turner 1982], Orwell [Wadler 1985], Miranda[1] [Turner 1986] and Haskell [Hudak and Wadler 1990]. Its a lazy strongly typed combinator languages (i.e. it has no lambda abstractions). Programs are called scripts and the things one can define in scripts are: functions, conformals, types (from other types) and constructors (as part of a new sum type). An example of a small BAWL script can be seen in Figure 28.

```
fac 0 = 1
fac n = n*fac(n-1)
fib n = 1,                  if n<=1
      = fib(n-1)+fib(n-2), otherwise
primes = sieve [2..]
         where
           sieve (p:x) = p:sieve[n|n<-x;n mod p>0]
```

Figure 28: Example of BAWL *Script*

In this example, `fac` is the factorial function, `fib` a naive version of the fibonacci function and `primes` a conformal defined to be the infinite list of primes, calculated by the method called "The Sieve of Eratosthenes".

As can be seen from the example, expressions can contain literals (constants; e.g. `0`,`1`,`'a'`,etc.), variables, constructors (`:`,`[]`,...), operators (`+`,`-`,`++`,...), applications, list comprehensions and arithmetic sequences. The languages also supports tuples.

Arithmetic sequences are expressions denoting sequences of integers. Examples could be:

```
nats     = [1..]
evennats = [2,4..]
```

List comprehension is a strong new language feature included in many new functional languages. It is included in among other KRC, Miranda and Haskell. List comprehensions are related is to set comprehensions in mathematics. An example could be:

```
oddnats = [x|x<-nats; odd x]
```

This definition reads: let `oddnats` be the `x`'s taken from `nats` for which `x` is odd.

[1] Miranda is a trademark of Research Software Ltd.

Patterns may include literals, variables, constructor patterns and tuples. Pattern matching in function definitions are from left to right, both in sequences of patterns, alternatives and equations. The boolean constructors `True` and `False` and the list constructors `:` and `[]` are the only constructors predefined in the initial constructor environment.

10.1 Conformals

We will now explain what conformals are. Conformals are global (constants) structures defined by:

conformal ::= *pat* = *rhs*

where *pat* is a pattern and *rhs* a right hand side defining a value that the pattern is matched against. The syntax of right hand sides will be discussed in the next section. This means that functions of arity zero are also conformals. An example could be:

```
(fac10,fib10) = (fac 10,fib 10)
```

As we will see later conformals have to be treated specially to obtain lazy evaluation. We will not interpret general conformals like the ones shown above, but only the one that are zero arity functions. General conformals will be translated into our language by a simple transformation:

```
$p = rhs
x = $selx $p
$selx pat = x
...
```

where there is a selector `$selx` and a definition `x = $selx $p` for for every variable `x` in *pat*. The prefix `$` is used to make sure that variable do not clash with user defined names.

10.2 The Syntax of BAWL

This section describes the syntax of BAWL. Figure 29 shows the syntax of the language BAWL that the compiler is able to handle. The description is deliberately kept in a form that should be short and readable. Some parentheses may be omitted e.g. in applications, the guard may be omitted in case there is only one alternative, and in the last alternative **if** `True` may be replaced by **otherwise**. In scripts, definitions and alternatives may be separated by newline instead of semicolon and offset rules like those of Miranda holds for the scope of `where`'s. Local scripts may not contain type definitions. The predefined basic types are: num, real, string, char and bool.
Some language features normally present in related languages, but not in BAWL, are:

- Input
- Modules
- Plus patterns (e.g. `p+42`) and `as` patterns

10.3 Abstract Syntax

The syntax presented in the last section is the concrete syntax of BAWL, that is, the syntax that we may write scripts in. In the next chapter we will present the semantics of BAWL and there we will need an abstract syntax of BAWL.

scr	::=	*def*{;*def*}	(script)
def	::=	*fun eq*	(function definition)
	|	*pat* = *rhs*	(conformal definition)
	|	*tname* {*tvar*}::= *cdef*{| *cdef*}	(type declaration)
	|	*tname* {*tvar*} == *texpr*	(type alias)
cdef	::=	*con* {*atexpr*}	(constructor declaration)
texpr	::=	*atexpr*	(type expression)
	|	*tname* {*atexpr*}	(type construction)
	|	*texpr* -> *texpr*	(function type)
atexpr	::=	*tname*	(type name)
	|	*tvar*	(variable)
	|	(*texpr*$_1$,...,*texpr*$_n$)	(tuples, n=0 or n≥2)
	|	[*texpr*$_1$,...,*texpr*$_n$]	(lists, n≥0)
eq	::=	{*pat*} *rhs*	(equations)
rhs	::=	*alt*{; alt} {where *scr*}	(right hand side)
alt	::=	= *expr*, if *guard*	(alternatives)
	|	= *expr*, otherwise	(last alternative)
guard	::=	*expr*	(guard)
pat	::=	*lit*	(literal)
	|	*var*	(variable)
	|	(*pat*$_1$: *pat*$_2$)	(pair)
	|	[*pat*$_1$,...,*pat*$_n$]	(lists, n≥0)
	|	(*con* {*pat*})	(constructor pattern)
	|	(*pat*$_1$,...,*pat*$_n$)	(tuples, n=0 or n≥2)
expr	::=	*lit*	(literal)
	|	*var*	(variable)
	|	*fun*	(function or conformal)
	|	*con*	(constructor)
	|	(*expr*$_1$ *op* [*expr*$_2$])	(operator application
	|	(*op* [*expr*])	or sections)
	|	(*expr*$_1$:*expr*$_2$)	(pair)
	|	[*expr*$_1$,...,*expr*$_n$]	(lists, n≥0)
	|	(*expr*$_1$,...,*expr*$_n$)	(tuples, n=0 or n≥2)
	|	(*expr*$_1$ *expr*$_2$)	(application)
	|	[*expr*$_1$ [,*expr*$_2$] .. [*expr*$_3$]]	(arithmetic sequences)
	|	[*expr*| *qual* {;*qual*}]	(list comprehensions)
qual	::=	*expr*	(filter)
	|	*pat*<-*expr*	(generator)
op	::=	+|-|*|/|div|mod|^|=|~=|>|<|<=|>=|~|&|\/|#|!|++|--|.	

Figure 29: Syntax of BAWL

Figure 30 shows the syntax domains and their associated variables. The convention is that the syntax domains and variables have names corresponding to the nonterminals of the syntax and that lower case names are variables and the corresponding upper case names are their domains, e.g. con is a constructor name in the syntax domain Con. Syntax variables like for instance *def** range over syntax phrases of the form {*def* }.

Syntax variables:

scr ∈ Scr	(scripts)
def ∈ Def	(definitions)
tdef ∈ Tdef	(type definition)
fdef ∈ Fdef	(function definition)
cdef ∈ Cdef	(constructor declaration)
texpr ∈ Texpr	(type expression)
atexpr ∈ Atexpr	(argument type expression)
tname ∈ Tname	(type name)
tvar ∈ Tvar	(type variable)
eq ∈ Eq	(equations)
rhs ∈ Rhs	(right hand side)
alt ∈ Alt	(alternative)
guard ∈ Guard	(guard expression)
pat ∈ Pat	(pattern)
expr ∈ Expr	(expression)
lit ∈ Lit	(literal)
var ∈ Var	(variable)
fun ∈ Fun	(function name)
con ∈ Con	(constructor name)
op ∈ Op	(operator)
tpexpr ∈ Tpexpr	(tuple expression)
lcexpr ∈ LCexpr	(list comprehension)
qual ∈ Qual	(qualifier)
asexpr ∈ ASexpr	(arithmetic sequences)

Figure 30: Syntax Domains

There is little difference between the abstract syntax and the concrete syntax presented in this chapter and we will therefore not show the abstract syntax here. In the abstract syntax: all right hand sides will have a guard (the empty guard corresponds to the guard `True`); functions definitions of one function is merged into one, so we have *fdef* = *fun eq**; we will still use mix-fix notation productions like "`=` *expr*", **if** *guard* to make the abstract syntax more readable; etc. Productions with ... in them, like the one for tuples, are in fact infinite families of productions and can not be written directly in the abstract syntax. The production for list is handled by the parser, since `[`$expr_1$,...,$expr_n$`]` is just an abbreviation of `[`$expr_1$`,[`$expr_2$,...,$expr_n$`]...]`, but the production for tuples must be treated differently. We introduce two new names `Te` and `Tp` different from all names in (Con + Fun + Var + Op) and use these to construct tuple expressions and tuple patterns in our abstract syntax. Otherwise the abstract syntax is straightforward.

Chapter 11

Denotational semantics of BAWL

In this chapter we give a formal description of the semantics of the BAWL language implemented by the interpreter. The description is given as a denotational semantics using the style defined in Chapter 4. We assume that scripts and expressions are well typed. The denotation of an expression where evaluation results in an error is $\perp$. The semantics does not focus on laziness, that is, on sharing. We leave the question of how to make the evaluation lazy and how to print the result until next chapter where we describe the interpreter. The semantic definition in this chapter forms a basis for the development of the interpreters. In the end of the chapter, we will discuss the differences between the semantics of BAWL and Miranda. We will not discuss the static semantics of BAWL, since it is not the important issue of this report how to type programming languages like BAWL. This has been done elsewhere, see e.g. Peter Hancock's chapters in [Peyton Jones 1987].

11.1 Domains

The syntax domains and variables were described in Chapter 10. In this section we define the semantic domains used in the semantics. Figure 31 shows the domains, their associated variable and operations. Most of these domains and operations have a straightforward interpretation. The only somewhat strange operation is fcl, which is used to create function values. The effect of fcl is very close to the function usually called curry. It takes an integer n and a function f as input and returns the function:

$\lambda x_1 ... \lambda x_n .f\ [x_1,...,x_n]$

The purpose of fcl is to create function values corresponding to the BAWL functions defined in a script.

11.2 The Semantics of BAWL Scripts

Usually when we assign meaning to a program we do this by specifying the function that the program computes. This is done under the assumption that the program has some sort of main function whose meaning is the meaning of the entire program. But often one also wants to run programs in an interpreted environment (for development and debugging), and in this case it seems better to assign another meaning to a program. The meaning of a BAWL program, from now on called script, is therefore a function environment mapping function names to their meaning and a constructor environment mapping constructor names to their meaning. Figure 32 shows this formally[1].

[1] All lambdas in this chapter that are not explicitly shown, i.e. those on the left hand side of equations, are assumed strict. This will become important when we discuss translating the semantics into a Scheme interpreter.

Semantic algebra:

Domains:

$v,u \in \text{Value} = (\text{Basic} + \text{List} + \text{Tuple} + \text{Construct} + \text{Function})_\perp$
$\text{Basic} = \text{Integer} + \text{Real} + \text{Character}$
$l,m,n \in \text{Num} = \text{Integer} + \text{Real}$
$vs,us \in \text{List} = \text{Nil} + \text{Pair}$
$\text{Nil} = \text{Unit}$
$\text{Pair} = \text{Value} \times \text{List}$
$\text{Bool} = \text{True} + \text{False}$
$\text{True} = \{([\![\text{True}]\!],())\}$
$\text{False} = \{([\![\text{False}]\!],())\}$

Operations:

$[]: \text{Unit} \to \text{List}$
$[] = \text{inNil}()$

$(\_:\_): \text{Value} \times \text{List} \to \text{List}$
$(u{:}us) = \text{inPair}(u,us)$

$\text{null}: \text{List} \to \text{Bool}$
$\text{null}\ [] = \text{true}$
$\text{null}\ (u{:}us) = \text{false}$

Domains:

$\text{Tuple} = \Sigma_{n=0}^{\infty}\text{Value}^n$

Operations:

$\text{tupleToList}: \text{Tuple} \to \text{List}$
$\text{tupleToList}\ () = []$
$\text{tupleToList}\ (u,v) = u{:}\text{tupleToList}\ v$

$\text{listToTuple}: \text{List} \to \text{Tuple}$
$\text{listToTuple}\ [] = ()$
$\text{listToTuple}\ (u{:}us) = (u,\text{listToTuple}\ us)$

Domains:

$\text{Construct} = \Sigma_{n=0}^{\infty}(\text{Con} \times \text{Value}^n)$
$\text{Function} = \text{Value} \to \text{Value}$
$\rho \in \text{VEnv} = \text{Var} \to \text{Value}$ (variable environment)
$\gamma \in \text{CEnv} = \text{Con} \to \text{Value}$ (constructor environment)
$\phi \in \text{FEnv} = \text{Fun} \to \text{Value}$ (function environment)

Operations:

$\text{fcl}: \text{Integer} \to (\text{List} \to \text{Value}) \to \text{Value}$
$\text{fcl}\ n\ \kappa = n{=}0 \to \kappa\ []\ []\ \lambda v.\text{fcl}\ (n{-}1)\ (\underline{\lambda}vs.\kappa\ (v{:}vs))$

Figure 31: Semantic Domains

The initial function environment maps function names from the standard environment to their meanings. The initial constructor environments contains definitions of predefined constructors, in our case only `True` and `False`. The definition of both of these environments and the initial variable environment are straightforward and therefore omitted.

Valuation function:

S: Scr → (CEnv × FEnv)

$\mathbf{S}[\![scr]\!] = (\gamma, \mathbf{FE}[\![scr]\!]\rho_{init}\gamma\phi_{init})$

where

$\gamma = \mathbf{CE}[\![scr]\!]$

Figure 32: Semantics of BAWL *Scripts*

11.3 The Constructor Environment

The meaning of a constructor name depends only on the arity of the constructor and the constructor name itself. We take the meaning of a constructor to be a function of same arity as the constructor. This function maps values into an element in Construct with the constructor as the first component. The operation fcl builds this function. This operation expects its second argument to be continuation that it can apply to a list of values and this list is therefore converted to a tuple by the operation listToTuple.

The constructor environment is an environment mapping constructor names to their meanings. Figure 33 shows how to get a constructor environment from a script.

Valuation functions:

CE: Def* → CEnv

$\mathbf{CE}[\![(tname\ tvar^*\ cdef^*)def^*]\!] = \mathbf{CD}[\![cdef^*]\!]\ (\mathbf{CE}[\![def^*]\!])$

$\mathbf{CE}[\![fdef\ def^*]\!] = \mathbf{CE}[\![def^*]\!]$

$\mathbf{CE}[\![ ]\!] = \gamma_{init}$

CD: Cdef* → CEnv → CEnv

$\mathbf{CD}[\![(con\ atexpr^*)cdef^*]\!]\gamma =$

$[[\![con]\!] \mapsto \text{fcl (length}[\![atexpr^*]\!])\ (\underline{\lambda}\text{vs.}([\![con]\!], \text{listToTuple vs}))](\mathbf{CD}[\![cdef^*]\!]\gamma)$

$\mathbf{CD}[\![ ]\!]\gamma = \gamma$

Figure 33: The Semantics of the Constructor Environment

The valuation function **CE** takes a list of definitions and returns a constructor environment. If a definition is a function definition it is skipped. If it is a type definition then the valuation function **CD** is called with the list of constructor definitions from the type definition and an environment which is the result of evaluating **CE** on the rest of the definitions. **CD** then build up a constructor environment from the list of constructor definitions and a constructor environment. The semantics of the constructor environment is shown in Figure 33.

11.4 The Function Environment

The meaning of a function is a mapping from values to a value. In BAWL functions are defined in the script and the construction of a function environment (mapping function names to function values) therefore depends only on the script.

Figure 34 shows semantic functions giving meaning to function definitions in a script. The function **FE** takes a script, a variable environment (if the definitions are local, there might be free variables in these), a constructor environment and a function environment as input, and returns a function environment.

Valuation functions:

FE: Scr → VEnv → CEnv → FEnv → FEnv
$\mathbf{FE}[\![scr]\!]\rho\gamma\phi_1 = \text{fix}(\lambda\phi.\underline{\lambda}\text{f}.\mathbf{FD}[\![scr]\!]\text{f}\rho\gamma\phi\phi_1)$

FD:Def* → Fun → VEnv → CEnv → FEnv → FEnv → Value
$\mathbf{FD}[\![(fun\ eq^*)def^*]\!]\text{f}\rho\gamma\phi\phi_1 =$
$\quad \text{f}=[\![fun]\!] \rightarrow \text{fcl}\ (\text{funArity}[\![eq^*]\!])\ (\mathbf{ME}[\![eq^*]\!]\rho\gamma\phi) \square \mathbf{FD}[\![def^*]\!]\text{f}\rho\gamma\phi\phi_1$
$\mathbf{FD}[\![tdef\ def^*]\!]\text{f}\rho\gamma\phi\phi_1 = \mathbf{FD}[\![def^*]\!]\text{f}\rho\gamma\phi\phi_1$
$\mathbf{FD}[\![\,]\!]\text{f}\rho\gamma\phi\phi_1 = \phi_1\ \text{f}$

ME:Eq* → VEnv → CEnv → FEnv → List → Value
$\mathbf{ME}[\![(pat^*\ alt^*\ \textbf{where}\ def^*)eq^*]\!]\rho\gamma\phi\text{vs} =$
$\quad \mathbf{MP}[\![pat^*]\!]\rho(\underline{\lambda}\rho_1.\mathbf{MA}[\![alt^*]\!]\rho_1\gamma(\mathbf{FE}[\![def^*]\!]\rho_1\gamma\phi))(\mathbf{ME}[\![eq^*]\!]\rho\gamma\phi\text{vs})\text{vs}$
$\mathbf{ME}[\![\,]\!]\rho\gamma\phi\text{vs} = \bot$

MA:Alt* → VEnv → CEnv → FEnv → Value
$\mathbf{MA}[\![(\texttt{=}\ expr, \textbf{if}\ guard)alt^*]\!]\rho\gamma\phi = \mathbf{E}[\![guard]\!]\rho\gamma\phi \rightarrow \mathbf{E}[\![expr]\!]\rho\gamma\phi \square \mathbf{MA}[\![alt^*]\!]\rho\gamma\phi$
$\mathbf{MA}[\![\,]\!]\rho\gamma\phi = \bot$

Figure 34: The Semantics of the Function Environment

The function **FD** assigns meaning to a function name f when given a sequence of definitions, a variable environment, a constructor environment and two function environment. If the first definition is a function definition, then **FD** tests whether or not f is equal to the function being defined and if so it builds up a lambda expression by calling fcl:

$\text{fcl}\ (\text{funArity}[\![eq^*]\!])\ (\mathbf{ME}[\![eq^*]\!]\rho\gamma\phi)$

The syntax operation funArity finds the arity of the function. It does so by examining the equations and as BAWLis defined here the arity of a function is equal to the length of the sequence of patterns in the equations. The second argument to fcl, the continuation, is eventually applied to a list of value arguments, when the function has been applied to all its arguments. If f is not equal to the function being defined or the definition is a type definition then the definition is skipped. Finally, if there are no more definitions the initial function environment is applied to f.

ME matches the patterns of the equations against a list of values. It does this by calling **MP** with a success continuation and a failure continuation. If the match succeeds, the success continuation is applied to the variable environment resulting from the matching. If the match fails, the failure continuation is called and the next equations will be matched. **MP** will be described later.

MA takes a list of alternatives, the usual environments and it finds the first alternative with a guard that evaluates to true. When such a guard is found **MA** evaluates the expression of the alternative. Again if there is no more alternatives $\bot$ is returned.

11.5 The Semantics of Expressions

In this section we describe and give a definition of the semantics for expressions.

The semantics of expressions is given in Figure 35. The valuation function for expressions is **E** which besides an expression takes a variable environment, a function environment and a

constructor environment as input and returns a value. Most of the rules for **E** are straightforward and will not be described in detail. The function **L** gives meaning to literals and the function **B** gives meaning to basic operators (`+,-,*,#,++,...`). For example the meaning of `+` could be:

$$\mathbf{B}[\![\texttt{+}]\!] = \lambda v_1.\lambda v_2.v_1 + v_2$$

The definition of **L** and **B** are straightforward and therefore omitted.

The functions **AS** and **LC** give meaning to arithmetic sequences and list comprehensions and are treated later.

Valuation functions:

E: Expr → VEnv → CEnv → FEnv → Value

$$\begin{aligned}
&\mathbf{E}[\![lit]\!]\rho\gamma\phi && = \mathbf{L}[\![lit]\!]\\
&\mathbf{E}[\![var]\!]\rho\gamma\phi && = \rho[\![var]\!]\\
&\mathbf{E}[\![fun]\!]\rho\gamma\phi && = \phi[\![fun]\!]\\
&\mathbf{E}[\![con]\!]\rho\gamma\phi && = \gamma[\![con]\!]\\
&\mathbf{E}[\![op]\!]\rho\gamma\phi && = \mathbf{B}[\![op]\!]\\
&\mathbf{E}[\![\texttt{[]}]\!]\rho\gamma\phi && = []\\
&\mathbf{E}[\![\texttt{(}expr_1\texttt{:}expr_2\texttt{)}]\!]\rho\gamma\phi && = \mathbf{E}[\![expr_1]\!]\rho\gamma\phi : \mathbf{E}[\![expr_2]\!]\rho\gamma\phi\\
&\mathbf{E}[\![\texttt{(Te }expr^*\texttt{)}]\!]\rho\gamma\phi && = \text{listToTuple } (\mathbf{E*}[\![expr^*]\!]\rho\gamma\phi)\\
&\mathbf{E}[\![\texttt{(}expr_1\ expr_2\texttt{)}]\!]\rho\gamma\phi && = \mathbf{E}[\![expr_1]\!]\rho\gamma\phi(\mathbf{E}[\![expr_2]\!]\rho\gamma\phi)\\
&\mathbf{E}[\![asexpr]\!]\rho\gamma\phi && = \mathbf{AS}[\![asexpr]\!]\rho\gamma\phi\\
&\mathbf{E}[\![lcexpr]\!]\rho\gamma\phi && = \mathbf{LC}[\![lcexpr]\!]\gamma\phi\rho\ []
\end{aligned}$$

E: Expr* → VEnv → CEnv → FEnv → List

$$\begin{aligned}
&\mathbf{E*}[\![expr\ expr^*]\!]\rho\gamma\phi && = \mathbf{E}[\![expr]\!]\rho\gamma\phi : \mathbf{E*}[\![expr^*]\!]\rho\gamma\phi\\
&\mathbf{E*}[\![\,]\!]\rho\gamma\phi && = []
\end{aligned}$$

Figure 35: Semantics of expressions

11.5.1 The Semantics of Arithmetic Sequences

Figure 36 shows the semantics of arithmetic sequences and contains no surprises. The operations from, fromto ... all operates on numerals (num) values and returns lists of numerals.

11.5.2 The Semantics of List Comprehensions

The the semantics of list comprehensions that we present in this section is given in a very direct way. We could have chosen just to describe a transformation of list comprehensions into equivalent expressions without list comprehensions. This is what the Haskell report suggests [Hudak and Wadler 1990]. But since BAWL does not have lambda abstractions this has the disadvantage that we would have to introduce new local function definitions (giving name-capture problems). This is not such a serious problem, more serious is the fact that translating

`[`*expr* `|x<-`*expr′*`;`*qual*∗`]`

into

Semantic algebras:

Operations

from: Num → List
from n = n : from (n+1)

fromThen, fromTo: Num → Num → List
fromThen n l = n : fromThen (n+l)
fromTo n m = n≤m → n : fromTo (n+1) m [] []

fromUpTo, fromDownTo: Num → Num → Num → List
fromUpTo n m l = n≤m → n : fromUpTo (n+l) m l [] []
fromDownTo n m l = n≥m → n : fromDownTo (n+l) m l [] []

Valuation functions:

AS: ASexpr → VEnv → CEnv → FEnv → List

$$\mathbf{AS}[\![\texttt{[}expr\texttt{..]}]\!]\rho\gamma\phi = \text{from } (\mathrm{E}[\![expr]\!]\rho\gamma\phi)$$

$$\mathbf{AS}[\![\texttt{[}expr_1\texttt{,}expr_2\texttt{..]}]\!]\rho\gamma\phi = \text{fromThen } l_1\ (l_2{-}l_1) \text{ where } l_i = \mathrm{E}[\![expr_i]\!]\rho\gamma\phi$$

$$\mathbf{AS}[\![\texttt{[}expr_1\texttt{..}expr_2\texttt{]}]\!]\rho\gamma\phi = \text{fromTo } (\mathrm{E}[\![expr_1]\!]\rho\gamma\phi)\ (\mathrm{E}[\![expr_2]\!]\rho\gamma\phi)$$

$$\mathbf{AS}[\![\texttt{[}expr_1\texttt{,}expr_2\texttt{..}expr_3\texttt{]}]\!]\rho\gamma\phi = l_1{\le}l_2 \rightarrow \text{fromUpTo } l_1\ l_3\ l \square \text{fromDownTo } l_1\ l_3\ l \text{ where } l_i = \mathrm{E}[\![expr_i]\!]\rho\gamma\phi,\ l = l_2{-}l_1$$

Figure 36: Semantics of arithmetic sequences

```
concat (map f expr')
where
 f x = [expr|qual*]
```

gives code that first construct a list of lists (`(map f ...)`) and then concatenates the lists into one list, i.e. it constructs an intermediate list. This is not the case in in the semantics that we are about to present and a compiler generated from this semantics will reflect this and therefore be "efficient". But maybe the most important argument for treating list comprehensions as we do is that the whole point of the thesis is that we can generate translators by writing interpreters. The semantics of list comprehensions that we present may look complicated, but so does a "efficient" translation scheme for list comprehensions to (see for example [Peyton Jones 1987] Chapter 7[2]).

To explain the valuation functions for list comprehensions consider the following example:

```
[expr|x<-[1..];qual*]
```

this is equivalent to:

```
([expr|qual*] where x=1)++[expr|x<-[2..];qual*]
```

where we have used a somewhat sloppy BAWL notation. If *qual** is empty then `[`*expr*`|`*qual**`]` means `[`*expr*`]`. This gives an idea about how to interpret list comprehensions. First we give

[2]It does in fact turn out, though this is a coincidence, that the translation of list comprehensions that we generate from our semantics is equivalent to the one presented in [Peyton Jones 1987].

meaning to `([`$expr$`|`$qual*$`] where x=1)` remembering that we later have to append the meaning of `[`$expr$`|x<-[2..];`$qual*$`]` to this. This means that the meaning of a list comprehension `[`$expr_0$`|`$qual*$`]` will be given by:

$$\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}qual*\texttt{]}]\!]\gamma\phi\rho\ []$$

where [] is what we have to append to the meaning of `[`$expr_0$`|`$qual*$`]` from the start. If the first qualifier is a generator pat`<-`$expr_1$ the function **G** is applied to the pattern pat; a continuation, that when given a variable environment and a list, evaluates the rest of the list comprehension like above; the variable environment; the value vs to append and the result of evaluating $expr_1$. This gives the following rule:

$$\begin{aligned}&\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}pat\texttt{<-}expr_1\texttt{;}qual*\texttt{]}]\!]\gamma\phi\rho\mathrm{vs} = \\ &\quad \mathbf{G}[\![pat]\!](\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}qual*\texttt{]}]\!]\gamma\phi)\rho\mathrm{vs}(\mathbf{E}[\![expr_1]\!]\rho\gamma\phi)\end{aligned}$$

The rules for boolean qualifiers is straightforward and can be seen in figure 37. If the sequence of qualifiers is empty then a pair of the value of the expression in the list comprehension and the value vs is returned.

The function **G** match the pattern against each element in the list from the generator. If the list is empty then the value vs_1 is returned, but if the list is not empty, **G** matches the first element against the pattern. If the pattern matches the success continuation

$$\underline{\lambda}\rho_1.\kappa\ \rho_1\ (\mathbf{G}[\![pat]\!]\kappa\rho\mathrm{vs}_1\mathrm{us})$$

is applied to the environment resulting from the matching. This will be equivalent to something of the form:

$$\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}qual*\texttt{]}]\!]\gamma\phi\rho_1(\mathbf{G}[\![pat]\!]\kappa\rho\mathrm{vs}_1\mathrm{us})$$

This means that **LC** is applied to the rest of the list comprehension and a new vs to append later as in the example above. If the pattern does not match u then the failure continuation

$$\mathbf{G}[\![pat]\!]\kappa\rho\mathrm{vs}_1\mathrm{us}$$

is returned, which means that the element in the list is skipped. Figure 37 shows the all the rules of the semantics of list comprehensions.

The semantics for list comprehensions seems to violate the rule that says that denotational semantics shall be compositional. This is true if we read the definition is it is written, but we must remember that the notation

`[`$expr_0$`|`pat`<-`$expr_1$`;`$qual*$`]`

is really a short hand notation for an abstract syntax tree. This tree could look like:

$$\mathrm{lc}(qual_1,\mathrm{lc}(qual_2,(...,\mathrm{lc}(qual_n),\mathrm{expr})))$$

This means that `[`$expr_0$`|`$qual*$`]` becomes a proper subphrase of the list comprehension above.

Semantic algebras:

Domains

$\kappa \in \text{Cont} = \text{VEnv} \to \text{List} \to \text{List}$

Valuation functions:

$\mathbf{LC}$: LCexpr $\to$ CEnv $\to$ FEnv $\to$ VEnv $\to$ List $\to$ List

$\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}\,pat\texttt{<-}\,expr_1\texttt{;}\,qual\texttt{*]}\,]\!]\gamma\phi\rho\text{vs} =$
$\quad \mathbf{G}[\![pat]\!](\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}\,qual\texttt{*]}\,]\!]\gamma\phi)\rho\text{vs}(\mathbf{E}[\![expr_1]\!]\rho\gamma\phi)$
$\mathbf{LC}[\![\texttt{[}expr_0\texttt{|}\,expr_1\texttt{;}\,qual\texttt{*]}\,]\!]\gamma\phi\rho\text{vs} =$
$\quad \mathbf{E}[\![expr_1]\!]\rho\gamma\phi \to \mathbf{LC}[\![\texttt{[}expr_0\texttt{|}\,qual\texttt{*]}\,]\!]\gamma\phi\rho\text{vs} \,[]\, \text{vs}$
$\mathbf{LC}[\![\texttt{[}expr_0\texttt{|]}\,]\!]\gamma\phi\rho\text{vs} = \mathbf{E}[\![expr_0]\!]\rho\gamma\phi : \text{vs}$

$\mathbf{G}$: Pat $\to$ Cont $\to$ VEnv $\to$ List $\to$ List $\to$ List

$\mathbf{G}[\![pat]\!]\kappa\rho\text{vs}_1 =$
$\quad \underline{\lambda}\text{vs}.\mathbf{cases}\ \text{vs}\ \mathbf{of}$
$\qquad [] \quad : \text{vs}_1$
$\qquad (\text{u:us}) : \mathbf{P}[\![pat]\!]\rho(\underline{\lambda}\rho_1.\kappa\ \rho_1\ (\mathbf{G}[\![pat]\!]\kappa\rho\text{vs}_1\text{us}))\ (\mathbf{G}[\![pat]\!]\kappa\rho\text{vs}_1\text{us})\text{u}$

Figure 37: Semantics for list comprehension

Semantic algebras:

Domains

$\kappa_s \in \text{Succ} = \text{VEnv} \to \text{Value}$
$\kappa_f \in \text{Fail} = \text{Value}$

Valuation functions:

$\mathbf{MP}$:Pat* $\to$ VEnv $\to$ Succ $\to$ Fail $\to$ List $\to$ Value

$\mathbf{MP}[\![pat\ pat^*]\!]\rho\kappa_s\kappa_f = \underline{\lambda}(\text{v:vs}).\mathbf{P}[\![pat]\!]\rho(\underline{\lambda}\rho_1.\mathbf{MP}[\![pat^*]\!]\rho_1\kappa_s\kappa_f\text{vs})\kappa_f\text{v}$
$\mathbf{MP}[\![\,]\!]\rho\kappa_s\kappa_f = \underline{\lambda}[].\kappa_s\ \rho$

P: Pat $\to$ VEnv $\to$ Succ $\to$ Fail $\to$ Value $\to$ Value

$\mathbf{P}[\![var]\!]\rho\kappa_s\kappa_f = \lambda\text{v}.\kappa_s\ ([[\![var]\!] \mapsto \text{v}]\rho)$
$\mathbf{P}[\![lit]\!]\rho\kappa_s\kappa_f = \underline{\lambda}\text{v.v}=\mathbf{LP}[\![lit]\!] \to \kappa_s\ \rho\ []\ \kappa_f$
$\mathbf{P}[\![con\ pat^*]\!]\rho\kappa_s\kappa_f = \underline{\lambda}(\text{c,v}).\text{c}=[\![con]\!] \to \mathbf{MP}[\![pat^*]\!]\rho\kappa_s\kappa_f(\text{tupleToList v})\ []\ \kappa_f$
$\mathbf{P}[\![\texttt{(}pat_1\texttt{:}pat_2\texttt{)}\,]\!]\rho\kappa_s\kappa_f = \underline{\lambda}\text{vs}.\mathbf{cases}\ \text{vs}\ \mathbf{of}$
$\qquad\qquad [] \quad : \kappa_f$
$\qquad\qquad (\text{u:us}) : \mathbf{P}[\![pat_1]\!]\rho(\underline{\lambda}\rho_1.\mathbf{P}[\![pat_2]\!]\rho_1\kappa_s\kappa_f\text{us})\kappa_f\text{u}$
$\mathbf{P}[\![\texttt{[]}\,]\!]\rho\kappa_s\kappa_f = \underline{\lambda}\text{vs.null vs} \to \kappa_s\ \rho\ []\ \kappa_f$
$\mathbf{P}[\![\texttt{(Tp}\ pat^*\texttt{)}\,]\!]\rho\kappa_s\kappa_f = \lambda\text{v}.\mathbf{MP}[\![pat^*]\!]\rho\kappa_s\kappa_f(\text{tupleToList v})$

Figure 38: Semantics of patterns

11.6 Semantics of Pattern Matching

Now we are ready to present the rules for pattern matching. Patterns are matched against values and the result of this is an environment binding the pattern variables to corresponding parts of

the value. A match can also fail and in this case the result would be some value representing failure. What we usually try to do is to match alternative sequences of patterns. Then a failure to match is not an error, but an indication that we should try the next sequence of patterns. This is one reason for using continuation passing style in the semantics definition for patterns. Another reason has to do with partial evaluation of the interpreter and is described in Chapter 13. Figure 38 shows the semantics of pattern matching.

The valuation function **MP** matches a sequence of patterns against a list of values, where all patterns have to match. It does this by first matching the first pattern and if this matching succeeds, it matches the rest of the patterns. The first rule for **MP** therefore has the form:

$$\mathbf{MP}[\![pat\ pat^*]\!]\rho\kappa_s\kappa_f = \underline{\lambda}(\text{v:vs}).\mathbf{P}[\![pat]\!]\rho(\lambda\rho_1.\mathbf{MP}[\![pat^*]\!]\rho_1\kappa_s\kappa_f\text{vs})\kappa_f\text{v}$$

where the success continuation contains a call to **MP** with the rest of the patterns $pat*$. If the sequence of patterns is empty then the whole matching has succeeded and the initial success continuation κ_s (the one supplied to the original call to **MP**) can be applied to the environment and the result of this is then a function.

The valuation function **P** does the actual matching of a pattern against an argument. If the pattern is a variable the matching succeeds and the environment is updated with a binding of the variable to the argument. If the pattern is a literal this is mapped into the corresponding value by **LP** and then compared to the argument. If the test succeeds the success continuation is applied to the unchanged environment, otherwise the failure continuation is called.

The rules for pair patterns and constructor patterns are very close in form, which is not so strange since ":" is really just another constructor. So we will only explain the rule for constructor patterns. The matching of a constructor pattern (*con pat**) succeeds, if the value v has the same constructor *con* in its left component as the one in the pattern and if the right component of v matches the patterns *pat**, otherwise it fails. The matching of these patterns is again done by **MP**, but we first have to transform the tuple value into a list value. This is done by tupleToList. The rule for nil patterns is similar to the rule for literal patterns or an argumentless constructor.

Finally the rule for tuple patterns is also very similar to the one for constructor patterns. Since programs are assumed well typed we know that the value is a tuple of the right kind and it is not necessary to check that this tuple has the right arity. We will therefore not need the test like in the case of constructor patterns. So the right hand side of the rule for tuple patterns is the same as for constructor patterns except that we know that the test is always true.

There is one problem concerning pattern matching of constructors. If we have a function f that takes a tuple as an argument:

```
f (x,y) = ...
```

then matching will always succeed since we are working with well typed programs[3]. But what if we had a type with only one constructor like:

```
sometype * ** = C * **
```

and a function g of the form:

```
g (C x y) = ...
```

we could then expect that pattern matching for g behaves exactly as for f. This means that we have to treat pattern matching for these cases in a special way and this is in fact also what

[3]Patterns that behave like this are sometimes called *irrefutable*.

Miranda does. We will however not do so in our semantics, in order not to complicate it to much.

A point in which the described semantics differ from the implementation, is that the interpreter can handle non-linear patterns, i.e. that have multiple occurrences of the same variable in one equation. Matching succeeds only if all occurrence the variable are bound to the same value.

11.7 Running BAWL programs

The meaning that **S** assigns to BAWL scripts is not the standard one, that is, a function from input to output, but two environments that assigns meaning to function names and constructor names in the script. We may, however, easily do so quite simple. If *fun* is the name of some function defined in a script *scr*, then

$$\textbf{snd}\ (\mathbf{S}[\![scr]\!])\ [\![\text{fun}]\!]$$

is the function that *fun* computes. The function name *fun* could then be either a fixed name, e.g. `main` or some name specified in the script by some convention, e.g. the first or the last function.

We can also specify the meaning of an expression with respect to a script. This is shown in Figure 39. This corresponds to having a developing and debugging environment in which the user can enter any expression containing function and constructor names defined in a given script.

Valuation function:

$$\begin{aligned}&\mathbf{I}\text{: Scr} \rightarrow \text{ Expr } \rightarrow \text{ Value}\\ &\mathbf{I}[\![scr]\!][\![expr]\!] = \mathbf{E}[\![expr]\!]\rho_{init}\gamma\phi\\ &\qquad\qquad\textbf{where}\\ &\qquad\qquad(\gamma,\phi) = \mathbf{S}[\![scr]\!]\end{aligned}$$

Figure 39: Interpretation of BAWL *programs*

11.8 Differences Between Miranda and BAWL

In this section we will give a summary of the semantic differences between Miranda and BAWL. When we refer to the semantics of Miranda we will mean what we have been able to discover from the operational behaviour of the Miranda System, since we are not aware of any published formal semantics of Miranda.

Here are the differences that we have observed:

- BAWL starts pattern matching when a function is applied to as many arguments as given by the definition, while Miranda uses type information to decide when to start matching. The Miranda system write `<function>` when one asks it to evaluate the expression `f []` with the script (`const` is a standard function version of the K-combinator):

  ```
  f [] = bottom 0
  f x  = const
  bottom x = bottom x
  ```

- Matching of constructor from singular constructor families will force evaluation in BAWL, but not in Miranda. This will however not be the case in our implementation. There is however an inconsistency in the semantics of Miranda, since both `f` and `g` defined in the following script will evaluate their arguments.

```
f () = 1
sometype ::= C
g (C) = 1
```

- Miranda has non-linear patterns. This is not described in our semantics, but will be the case in our final implementation.

Chapter 12

The First Interpreter

This chapter describes the first simple interpreter. We will not describe the structure of the interpreter, as it follows the semantics very closely. What we will do instead is show the few points on which it deviate from the semantics. We also discuss the important issue of how to make the implementation lazy. The structure of the final interpreter will be discussed in the next chapter since this is derived from first interpreter by use of binding time improvements and by adding other optimizations. The text of interpreter can be found in Appendix C.

12.1 The Structure of the Interpreter

The interpreter was derived from the semantics by systematically applying a simple translation similar to that of Figure 9. The difference is that, where ever strict and non-strict functions are completely separated, we abstain from doing any translations for the strict functions and the corresponding applications. Strict functions that may be "mixed" with non-strict functions are coerced at their creation points. This means that the insertion of coercions is at least consistent (the corresponding definition is well typed) in the sense discussed in Section 5.3, but also minimal[1].

Using this simple translation gives a very good result, since there are very few applications that may apply both strict and non-strict functions. The "mixing" of strict and non-strict functions happens only because pattern matching may or may not evaluate arguments. Examining the semantics again, one may see by looking at the equations for the **P** valuation function, that the lambdas on the right side are in fact both strict and non-strict. Using the type inference method would not give a better result here, since the lambdas the lambdas are "mixed" immediately after their creation, which means that the coercions must be placed more or less as the simple translation places them.

A remark: We have performed some simplifications during the translation, such as uncurring the function generated when ever possible, unfolding of lets, etc. If we used the translation directly on the definition of **P** we would get a result that looked like:

[1] This would not be the case, had the equations defining **P** in the semantics been ordered differently.

```
(define P
  (lambda (pat)
    (lambda (venv)
      (lambda (sc)
        (lambda (fc)
          (cond
           ((PVar? pat) (lambda (v) ...))
           ((PLit? pat) (suspend-function (lambda (v) ...)))
           ...))))))
```

Instead we uncurry the function `P` and use the simple of the explicit coercion $\Uparrow$ that we described in section 5.6 that can be used when $\Uparrow$ is applied to a lambda abstraction. We then get a result of the form:

```
(define (P pat venv sc fc v)
  (cond
   ((PVar? pat) ...)
   ((PLit? pat) (let ((v (v))) ...))
   ...))
```

which is looks like the definition of `P` in Appendix C, except that some of the introduced let expression have been unfolded. **End of remark**

The interpreter takes a script and an expression `expr` as input and evaluates `expr` in the scope of the environments created by the function `S`. Since we divide the variable environment into two different ones (the variable environment and the function environment) we will also need a way to distinguish between variables to know which environment to use when looking up values. The information that we need for this is the names of the function in the script and we therefore let S also return this information. We will call the result from an application of `S` to a script an *info-structure*.

The interpreter `I` that we wish to partially evaluate has the following form:

```
(define (I scr expr)
  (let* ([info (S scr)]
         [fenv (vector-ref info 0)]
         [cenv (vector-ref info 1)]
         [funs (vector-ref info 2)])
    (print (E (elab-expr expr funs) (init-venv) cenv fenv))))
```

where `E` is the evaluation function for expressions, `print` is the print routine (we will discuss this later) driving the evaluation and `elab-expr` does some preprocessing of expressions (this will be discussed later). The function `S` takes a script and returns a function environment, a constructor environment and a list of function names. It looks like:

```
(define (S scr)
  (let* ([scr  (elab-scr scr)]
         [funs (collect-functions scr)]
         [cenv (CE scr)]
         [fenv (FE scr (init-venv) cenv (init-fenv))])
    (vector
      (make-env-static (collect-constructors scr) cenv (init-cenv))
      (make-env-static funs fenv (init-fenv))
      funs)))
```

The functions `collect-user-functions` collects function names in a script (only those defined at the top level). The function `collect-constructors` collect a the constructor names in the script. The function `make-env-static` is important for binding time reasons and will be described in Chapter 13. The function `elab-scr` preprocesses script in a fashion similar to `elab-expr` and will be described below.

The rest of the interpreter follows the structure of the semantics very closely and we shall only treat a few details here. Otherwise the reader is referred to the listing in Appendix C.

All the primitive operation (`Elit->lit`, `inVnil`,...) operating on data structures can be found in the adt-file in Appendix E along with definitions of the semantic operators (`fcl`, `from`,...).

The preprocessing function `elab-scr` translates conformal definitions into function definitions as described in Chapter 10, unfold string patterns (to lists of characters) and annotates variable name that are function function names with this information[2]. The function `elab-expr` is part of the definition of `elab-scr` and handle the translation of expressions.

The two cases where bottom appeared in the semantics have been translated into calls of the Scheme function `error`. This is reasonable since the two cases actually correspond to runtime errors. To be able to give sensible error messages an extra parameter `name` has been added to `ME` and `MA`.

literal values:		
	Integers	by integers
	Reals	by reals
	Characters	by characters
	Booleans	by booleans
list values:		
	pairs	by pairs (created by `cons`)
	nil	by the atom `nil`
Constructed values:		
	tuples	by vectors, the first component is the arity of the tuple the rest are the elements.
	constructs	by vectors, first component is the constructor name the rest are the arguments.
Function values:		
	Functions	by functions

Figure 40: Representation of BAWL*-values in Scheme*

12.1.1 Representation of Values

As discussed in Section 5.6 we have to choose some Scheme representation of our BAWL-values. From the point of view of the interpreter this representation should be hidden behind primitive operations. We do this by letting all references to values take place through primitive operations defined in our adt-file (see Appendix E). We make two exception to this principle concerning booleans and functions. We choose to represent booleans by booleans, since we do not want to generate code like:

```
(if (VBool->Bool (inVBool (((B '!equal) (x)) (y)))) ...)
```

[2]The parser is context free and does therefore not distinguish variable names that represents functions form variable names that represents formal parameters.

where we have to first turn the result of the test in a boolean in our representation and then for the sake of the conditional turn this value back to a scheme boolean again. In the final version of the interpreter we will make more exceptions of this kind.

Figure 40 shows how we represented BAWL-values in Scheme. Representing the BAWL-value in Nil by the atom `nil` make the representation of the empty list and False different. This would not have been the case if we had represented the empty list by the empty list in Scheme.

12.2 Generate a Compiler from the Interpreter

If we want to generate a compiler from the interpreter `I`, all we have to do is apply the compiler generator cogen of Similix to `I` and we are done. But there is a problem with this approach, since `expr` in `I` is dynamic (we only know the script when compiling) while expressions in the script are static. This will make the first argument to `E` dynamic since the binding time analysis of Similix is monovariant. This spoils the whole idea of compiler generation since we want `expr` in `S` to be static to get any specialization of `E` with respect to expressions from the script. We will therefore choose a slightly different approach: we will only apply cogen to a program without the function `I` and make `S` our goal function. This will give us a compiler that given a script produces a residual program with a specialized version `S-0` of `S`:

```
(define (S-0)
  (lambda (fn_0)
    (vector
      ...)))
```

That is, the compiled version of a script `scr` is a Scheme program with a function `S-0` that when called returns a the same result as `(S scr)`.

Now that we have removed the interpreter function `I` from the program we can define a slightly different version of `I` that will be used when running compiled scripts:

```
(define (I info expr)
  (let ([fenv  (vector-ref info 0)]
        [cenv  (vector-ref info 1)]
        [funs  (vector-ref info 2)])
  (print (E (elab-expr expr funs) (init-venv) cenv fenv)))
```

This takes a info-structure `info` instead of a script as before. This info-structure can be the result of a compilation or can be produced by applying the function `S` to a script. We can also see from this that the third element of the info-structure is actually needed, because when we no longer have the script we cannot elaborate (with `elab-expr`) the expression. We can now define a function `run` that takes an expression `expr` and the name of a file which holds a compiled script and evaluates the expression with respect to the compiled script:

```
(define (run expr file-name)
  (begin
    (load file-name)
    (I (S-0) expr)))
```

12.3 Lazy Evaluation

Often *call by need* is called *lazy evaluation*, since it postpone evaluation of arguments until these become needed. But we want more than call by need, that is, we want it to be lazy in more than

evaluation of arguments. Assume that we write the following version of the fibonacci function:

```
fib n = fl!n
        where
          fl = 1:1:[fl!n + fl!(n+1)|n<-[0..]]
```

Here `fl` is not an argument but a conformal, and we surely do not want to calculate the conformal `fl` every time it is used.

To summarize: We want our implementation to be lazy in two ways, evaluation of arguments and evaluation of conformals.

12.3.1 Achieving Call by Need

Let us first see how we can achieve call by need. Call by need can be viewed as an optimization of call by name. Our translation ensure that the evaluation order is call by name which means that an argument is evaluated only if it is needed, but every time it is needed. To achieve call by need we have to make sure that arguments are evaluated at most once. There are several ways to do this, but the one presented here (taken from [Rees and Clinger 1986])[3]) is both very simple to understand and very efficient to execute[4]. For delaying an argument we now use

```
(save (lambda () e))
```

instead of just `(lambda () e)`. The function `save` is defined by:

```
(define (save s)
  (let ([v '()] [tag #f])
    (lambda ()
      (begin
        (unless tag
                (set! v (s))
                (set! tag #t))
        v))))
```

In this way the argument expression is only evaluated if the tag is false and the first time the argument is evaluated the tag is set to true. This ensures that arguments are evaluated at most once. The expression evaluation function `E` then handles application like this:

```
(cond
  ...
 ((EApply? expr)
  ((E (EApply->expr1 expr) venv cenv fenv)
   (save (lambda () (E (EApply->expr2 expr) venv cenv fenv)))))
  ...)
```

12.3.2 Making Evaluation of Conformals Lazy

To handle lazy evaluation of conformals we have to use a slightly different approach than we used to achieve call by need. The `save` operation of last subsection introduces a new memory cell (the v and tag variables) where the result of evaluating the argument is saved the first time it is evaluated. This works because the saved evaluation is unique, i.e. it is the argument to a

[3] Compiling lazy languages by partial evaluation is introduced in Bondorf [Bondorf 1991b].

[4] I have compared several different versions that people at DIKU have come up with and this one performed best.

specific application. But evaluation of a conformal can be initiated many places in a program (reexamine the example in Section 12.3) so in order to get evaluation of the conformal shared, we have to save its value in a global memory cell. This is done in the following way: before doing the fixed point iteration in `S` we create new memory cells for all the conformals in the script (top level only) and the evaluation of the conformals are then saved in these using a slightly different version of `save` called `save-at`:

```
(define (save-at a s)
  (lambda ()
    (unless (cdr a)
            (set-car! a (s))
            (set-cdr! a #t))
    (car a)))
```

`save-at` saves the result of evaluating `s` in the cell pointed to by `a`.

12.4 The Print Routine

A normal order evaluation strategy means the result of interpreting an expression is its normal form if this exist. But we do not want to evaluate an expression all the way to normal and the print out its value, since then it might take very long before anything is written out and even worse it would make it impossible to write interactive programs. The normal way implementations of lazy languages work is to evaluate an expression to *weak head normal form*. This informally means that we evaluate an expression until as much of it is evaluated as is needed in order to be able to print some of it out. When an expression has been evaluated to weak head normal form the result is passed to the print routine which prints the part out that is available and if the expression is not in normal form yet the print routine restarts the evaluation of the substructures that are not in normal form. The cases where the expression is not normal form is when it evaluates to a list or a constructed value. In these case the print routine restarts the evaluation by evaluating the elements of the list or the substructures of the constructed value.

Since we have not written a type checker we run in to problems when writing a print routine for our interpreter. If the print routine is printing a list it need to know if the type of the elements in the list is `char` or not. If the type is `char` then the list is a string an should be printed as such, otherwise it should be printed as a list. This can only be decided in general by examining the type (e.g. the empty list should be printed `[]` while the empty string should be printed ``''``. Miranda also prints objects of function type as `<function>`, even if the evaluation of the object does not terminate. This can again only be decided from the type of the object. On the other hand if an object has type unit (the null tuple) then Miranda evaluates the object before printing `()`.

Since the current system is an experimental one it is reasonable that we take a short cut in writing the print routine. We choose to only print object when evaluation to weak head normal form terminates and to print strings as lists of characters (i.e. ``` ``foo'' ``` is printed `['f','o','o']`).

The print routine can be found in the adt-file in Appendix E.

Chapter 13

Binding Time Improvements of the Interpreter

The purpose of this chapter will be twofold. Firstly, it will serve as a description of the final version of the interpreter, except for those aspects that concerns binding time unrelated optimizations. These will be described in the next chapter. Secondly, it will give examples of binding time improvements needed in compiler generation by partial evaluation. It will be the second of these that will determine the structure of the chapter, since the overall structure of the final interpreter is the same as the first one.

13.1 Using Static Information about Dynamic Data

In Chapter 12 we claimed that the function `make-env-static` was important for binding time reasons. We will now explain why. In the absence of the `make-env-static` function, the environments returned by the function `FE` can be applied to both static and dynamic values and Similix will make the conservative choice that all argument to these environments are dynamic. This will make variable lookup a completely dynamic operation which is definitely not what we want. We can make sure that the arguments become static by using "The Trick" (as explained in Section 7.1) in a new disguise. Since we can decide from the script alone which function names and constructor names the environments may applied to, we can collect these names and use them to make the arguments to the environment static. The collection is done by the two functions: `collect-functions` and `collect-constructors`, and `make-env-static` then does the actual improvement:

```
(define (make-env-static namelist env envi)
  (lambda (name)
    (dynamic-lookup name namelist env envi)))

(define (dynamic-lookup name namelist env envi)
  (if (null? namelist)
      (envi name)
      (let ([s-name (car namelist)])
        (if (equal? name s-name)
            (env s-name)
            (dynamic-lookup name (cdr namelist) env envi)))))
```

What is happening is in fact that we compile the script with one function (or constructor) being the goal function at a time and with Similix ensuring that no function is compiled more than once. If we had decided to let the compiled version of a script be just the target function of

one of the functions in the script (as suggested in Section 11.7) we would not have needed "The Trick" here.

13.2 Continuation Passing Style

Continuation passing style is used several places in the interpreter. Some of these simply stems from the semantics which is written in continuation passing style at places. The rest of the places the cause is simply that we need to use functions that returns partially static results. There are two situation in which we need this and these are: for optimizing pattern matching (described in Section 13.4) and for the evaluation order optimization (described in Chapter 14).

13.3 Simulating Partially Static Data Structures

We could have simulated partially static data structures using the definitions in Figure 25, but it turns out that it is better to use environments, since this will make things more uniform (the pattern matching also use environments to represent partially static data structures). The only place in the interpreter, except for the pattern matching part, where we want to use the method is in the function `fcl`. This function built up a list of value corresponding to the arguments to a function of a constructor. The length of this list will always be static, but the elements in the list will be dynamic. We will only use the optimization on functions and not on constructors. This means that we will have to define a new version of `fcl` that we will use to build functions. This will be called `fcl-f` and is defined as:

```
(define (fcl-f n a sc)
  (if (= n a)
      (sc (init-oenv))
      (lambda (v)
        (fcl-f (add1 n) a
               (lambda (r) (sc (upd-d (inOarg n (inOTop)) v r)))))))
```

The old version of `fcl` will be used for constructors and will now be called `fcl-c`. The code `(upd-d (inOarg n (inOTop)) v r)` binds the occurrence (essentially the position in the list) of a value `v` in the list to the value itself. The strange way the occurrence is generated has to do with the way pattern matching is optimized. This will be explained in details in the next section.

Without this optimization the target function of `f` defined as `f x y = (x,y)` will look like:

```
(define (fcl-f-0 cfst_12)
  (lambda (v_5)
    (lambda (v_6)
      (let* ([vs_7 (list v_5 v_6)]
             [v_8 (car vs_7)]
             [vs_9 (cdr vs_7)]
             [v_10 (car vs_9)]
             [vs_11 (cdr vs_9)])
        (listtotuple
          (list (save (lambda () (v_8)))
                (save (lambda () (v_10)))))))))
```

With the optimization this now look like:

```
(define (fcl-f-0)
  (lambda (v_5)
    (lambda (v_6)
      (listtotuple
        (list (save (lambda () (v_5)))
              (save (lambda () (v_6))))))))
```

Notice that this corresponds to having used the kind of simple reductions discussed in Section 2.2. The result obtained so far can of course be improved further, but this is a different story which we will return to in the next chapter.

13.4 Pattern Matching

A considerable amount of work has been put into optimizing the compiling of pattern matching. This seem in general to be the case for implementations of program languages with pattern matching [Rothwell 1991]. The ideas used are generally the same as we used in [Jørgensen 1991]. We will not go in to details with this part of the interpreter, but shall look at some of the main ideas and extension to our earlier work.

The general idea used in the optimization of pattern matching used in [Jørgensen 1991] is that we during matching collect all the information that we obtain when we perform a test or decompose an argument. The information gained from a test is used to update a description of the argument that the test was performed on. This information may then be used later to eliminate tests. When we decompose an argument, to perform matching against one of its substructures, the value of the substructure is saved in an environment called the *occurrence environment*. If we ever match against the substructure again we reuse the value saved in the occurrence environment. This results in sharing of evaluation.

One of the main differences between the present work and [Jørgensen 1991] is that now the language is statically typed and has user defined sum types (i.e. with constructors) instead of being strict and untyped (dynamic typed). Working with a statically typed language means that we can optimize the pattern matching even further than it was done in [Jørgensen 1991]. For example, if we know that an argument to a function is of type list of something. Then after a failed test for the empty list, we know that the argument must be a pair. This may save us a test later. Using the same kind of argumentation we may leave out the test for the last constructor in a family of user defined constructors. To be able to do so we need to carry information about these type around during interpretation. This is the reason for the extra argument `ctab` to many of the function in the interpreter. The function `make-ctab` create the table `ctab` from a script and the information kept in `ctab` is lists of constructor families of every user defined type.

Variables are no longer bound to values during pattern matching, but to the corresponding occurrences. For this we use a new environment called the *variable occurrence environment*. This keeps as much static as possible during partial evaluation, since every dynamic value in an argument to a specialized call give rise to a formal parameter in the residual function. Binding variable to their value would then give rise to extra parameters that are really not needed. Using this representation introduces some new problems that need to be solved.

Firstly, after the pattern matching is over we have to construct the real variable environment. This is done by the function `make-new-venv`.

Secondly, we have to be careful not to force evaluation of argument more than necessary during pattern matching. The problem here is tuples (and constructors from singular families) since these patterns are irrefutable. The problem is the following: if we match a tuple pattern `(x,y)` then we can not bind the variables `x` and `y` to any value without forcing the evaluation of the argument we match against. We can temporarily solve the problem as we did in the first

interpreter by applying the function tupleToList to the argument `v` since this this changes `v` into:

```
(list (my-delay (tuple-ref (my-force v) 0))
      (my-delay (tuple-ref (my-force v) 1)))
```

which means that `v` is only forced when one of the components of `v` is needed. But in the final version of the interpreter we want to share values of substructures of arguments by keeping track of their values using an occurrence environment. The problem is solved by a function called `occ-lookup` that lookups the value of an occurrence without forcing more than necessary, and at the same time update the occurrence environment. This means that `occ-lookup` has to be written in continuation passing style. There is in fact two versions of `occ-lookup`, one that one that returns a fully evaluated value and on that returns a suspended value. The reason the we had to introduce two values were not this difference, but had to do with the resulting binding time value of calls to `occ-lookup`.

This is a general problem. Assume that we have a call `(f ... c1)` to a function defined by:

```
(define (f ... c)
  (if ...
     (c d)
     ...)
```

Then the binding time value of `(f ... c1)` is the same as binding time value of `(c1 d)` and in fact the least upper bound of this for all possible calls to `f`. So, if `f` is called one place where `c` gets bound to a function that returns a dynamic result, then all calls to `f` will return a dynamic result.

In the case of `occ-lookup` we want some calls to return an environment (a closure value) and other calls to return a value (a dynamic value) and we have therefore separated `occ-lookup` into two versions.

The occurrences that we used in [Jørgensen 1991] were simpler than the ones we use now and we will therefore present the new definition here:

Occ ::= `top` | (`head` *Occ*) | ((`tail` *Occ*) | (*Integer Occ*) | (`con` *Occ*) | `free`

The occurrence *n* `top` where n is an integer is the occurrence of the *n*'th argument to a function. The occurrences `head` *Occ* and `tail` *Occ* is the head and tail of the occurrence *Occ*. The occurrence *n Occ* is the *n*'th element in of the tuple or constructed value at *Occ* and `con` *Occ* is the name of the constructor at *Occ*. The occurrence `free` is a special occurrence given to global variable (only for locally defined functions) that are needed when the variable environment is built after pattern matching.

13.4.1 Sharing of Right Hand Sides

A problem that was solved in [Jørgensen 1991], but in a very unsatisfactory way was the so called code duplication problem. The problem is that during the compilation of pattern matching the target code for right hand sides of equation may be duplicated. Examine the following BAWL program[1]:

```
unwieldy [] [] = A
unwieldy xs ys = B xs ys
```

Compiling without the solution we are about to present, would give a target program that had the form:

[1] This example is taken from Wadler's chapter on pattern matching in [Peyton Jones 1987]

```
(define (unwieldy)
  (lambda (xs)
    (lambda (ys)
      (if (Vnil? xs)
          (if (Vnil? ys)
              A
              ((B xs) ys))
          ((B xs) ys)))))
```

where A and B is target code corresponding to `A` and `B`. If B is a large expression, the increase in size caused by the compilation process can become significantly large. The trick that solves the problem is inserting a specialization point to ensure sharing. This was done in [Jørgensen 1991] by changing the call unfold annotation of the interpreter manually. What we do now is to change the interpreter in such a way that specialization points will be inserted automatically[2]. If we want to insert a specialization point at some expression *expr*, we replace it with `(generalize0 (lambda () `*expr*`))`. Where `generalize0` is defined by:

```
(define (generalize0 c)
  (if #t c (generalize c)))
```

and `generalize` is a dynamic primitive defined by:

```
(defprim-dynamic 1 generalize
  (lambda (x) x))
```

This works in the intended way, because Similix inserts specialization points at every dynamic lambda expression and the primitive `generalize` will force its argument to become dynamic. The function `generalize0` works as `generalize`, since it is not a primitive operation and it evaluate to the identity function, Similix may post unfold the call to the residual calls to `generalize0` in case the call are not shared. So both the conditional in `generalize0` and the call to `generalize` will be removed during specialization (compilation).

The question that remains to be answered is where in the interpreter we are going to insert the specialization point. We have chosen to insert these before the evaluation of right hand side of equations starts as suggested in the example above. That is at calls to `MA` in the interpreter. This may give problems in combination with the local evaluation order optimization describe in the next chapter and we will return to this problem there.

13.5 Different Kinds of Environments

Since environments may have different binding time properties, it is a good idea to choose different representations, update-functions and lookup-functions[3]. The most important reason for keeping environments that have different binding time properties separated is to separate values with different binding times, but there are also other reasons. If an environment maps static names to static values we may use a first order representation (association lists) for the environment and use primitive operations to do update and lookup-operations. This is a good idea both because we give the specializer less code to work with and because we introduce less

[2]The idea is due to Bondorf and is also used in Similix

[3]It is also a good idea to use different update and lookup-functions even for environment that have the same binding time properties. The reason for this is that this will help Similix's closure analysis (The closure analysis is part of Similix's binding time analysis), which is sticky, to separate the closures belonging to the different environments.

closure values in the interpreter (the complexity of similix's closure analysis is at least $O(n^4)$). A example of such an environment is the environment, called `voenv` in the interpreter, that maps variable names to their occurrences. This was explained in [Jørgensen 1991].

If an environment maps static names to dynamic values it is better to use a higher order environment, since now the environment is a partially static structure and we want to resolve the bindings at specialization time (compile time). An example of such an environment is the usual variable environment.

Chapter 14

Optimizations Used by the Final Compiler

In this chapter we will describe what kind of optimizations (like the ones discussed in Chapter 8) we have implemented in the final version of the interpreter and thus what kind of optimizations the final version of the compiler performs. It is important to remember what we discussed in Chapter 8 about optimizations. Optimizations of the kind that we talk about here are usually not optimizations that makes the interpreter run faster, on the contrary most of the optimizations slows down interpretation considerable. The optimizations that we talk about makes the generated compilers generate optimized code, that runs faster and/or uses less space.

We will first look at the one on-line optimization that we have implemented and then at the local optimizations. We will not discuss any global optimizations since we have not implemented any.

14.1 On-line Optimizations

The only on-line optimization that we have implemented is one that results in sharing of force operations. Let us first look at an example to see what this is about. Consider the factorial function. This can be defined in BAWL as:

```
fac 0 = 1
fac n = n*fac(n-1)
```

This would be compiled into the following function without the optimization:

```
(define (fcl-f-0-3 sc_0)
  (lambda (v_1)
    (if (struct-equal? (v_1) 0)
        1
        (* (v_1)
           ((fcl-f-0-3 sc_0) (save (lambda () (- (v_1) 1))))))))
```

From this we can see that the variable `v_1` may be forced several times during one evaluation of the body of `fcl-f-0-3`. But, if we change the interpreter to perform the optimization the generated compiler would compile `fac` into:

```
(define (fcl-f-0-3 sc_0)
   (lambda (v_1)
      (let ([v_2 (v_1)])
         (if (equal? 0 v_2)
             1
             (* v_2
                ((fcl-f-0-3 sc_0) (save (lambda () (- v_2 1))))))))))
```

As can be seen from this example, the variable `v_1` is only forced once and the value is then bound to `v_2`. At all the places where `v_1` was originally forced there is now just an occurrence of the variable `v_2`.

The optimization work by keeping track of all variables that have been forced during evaluation and their values. When a variables value is forced, the value is saved in the variable environment and the fact that the variables value has been forced is recorded. If a variable whose value has been forced is ever used again it will therefore be the forced value that is found, when the value of the variable is looked up in the environment. The optimization does in fact correspond to a very simple form of common subexpression elimination as can be seen from the example.

We will now explain how the interpreter was changed in order to obtain a compiler that does the optimization. We change the interpreter as suggested above. This means that the result of evaluating an expressions is now three values: the value of the expression, a list of variable names whose values have been forced and a possible updated variable environment. Since the binding time value of these three results are respectively dynamic, static and closure, we have to rewrite evaluation of expressions into continuation passing style (as explained in Section 7.2) to be able to keep the list of variable and the environment from becoming dynamic. In the change version of the interpreter `E` now take two new arguments: the list of variables `fo` and the continuation `c`. Here is what the interpretation of variable is changed to:

```
...
((EVar? expr)
 (let ([var (EVar->var expr)])
   (if (member var fo)
       (c (venv var) fo venv)
       (let ([v (my-force (venv var))])
         (c v (cons var fo) (upd var v venv))))))
...
```

Here `var` is the name of the variable. If `var` is member of the list `fo`, it means that the value of `var` has been forced and can be found directly by applying the variable environment `venv` to `var`. Otherwise the value found in the environment has to be forced, the environment updated with the forced value for `var` and `var` added to the list of forced variable, before the continuation is called. We will not go further into details with the changes made to the interpreter, since rewriting it into continuation passing style is straightforward. However, variable may be forced already during pattern matching, which means that we also had to change the part of the interpreter, that does pattern matching, but only slightly.

Since Similix ensures that no computation is duplicated we can be sure that there will be a residual let-expressions, corresponding to the second let-expressions in the code interpreting variables, only if a variable is used more than once. If a variable is used exactly once Similix will unfold the let-expressions. This ensures that let-expressions will only appear in target program when they are actually needed, that is when a variable may be forced more than once during some computation.

It is of course also possible to do the described optimization using a local analysis to detect

the points where the force operations may be inserted. We have programmed a version of the interpreter that does this and it was quite complicated, since it included a local strictness analysis and an usages count analysis. The local method does however have one advantage over the on-line method, in that it does not introduce any new higher order structures (or in the Similix terminology: closures). Having many closures in a program makes Similix's closure analysis of the program slow, but even worse if a closure is an argument to a specialized call, new variables are introduced for each dynamic substructure of the closure. This may introduce a lot of superfluous parameters in residual programs (target programs).

14.2 Local Optimizations

We have only used one local optimization that is based on a not completely trivial analysis and that is the restricting of environments to reduce the number of formals in target functions. This will be treated in Section 14.2.2, but first we will shortly describe some simple local optimizations that we have implemented.

14.2.1 Separate Treatment of Special Syntactical Cases

All of the cases treated in this section can somehow be viewed as peephole optimizations, that is, simple local optimizations. The optimizations all corresponds to treating special syntactical cases in a special way. Here is a list of the most important syntactical cases that we treat separately:

- Equations with no local definitions. This means that there is no need to have a local call to `fix`.
- Equations with no local conformals. This means that there is no need to make new cells for conformals.
- Alternatives with a guard-expression that is just `True`. This comes either from cases where there is only one alternative or from an `otherwise` guard.
- Total applications of operations, e.g. `2+x` or `#xs`. This means that `2+x` is compiled into: `(+ 2 (my-force x))` instead of `(((B '+) (my-delay 2)) x)`. That is, the operations are in-lined.
- Tuple-expressions of length less than 4. Small tuples are used more frequently so its is a good idea to treat these more efficiently than tuples in general.
- Simple arguments to applications, etc. We will discuss this in detail below.

Most of these optimizations are programmed by quite simple changes to the original interpreter. The in-lining of operations is an exception to this, but the change is straightforward, even though it involved a lot of tedious programming resulting in about 130 lines of Scheme code.

The only optimization that is not quite trivial is the last one. If we look at some application *expr expr′* in BAWL, the first version of the interpreter will interpret this like:

```
((E (EApply->expr1 expr) venv cenv fenv)
 (my-delay (E (EApply->expr2 expr) venv cenv fenv))))
```

where `expr` is the application. If we generate a compiler from the first version of the interpreter this will translate the application into something like (*sexpr* `(save (lambda ()` *sexpr′*`)))` where

sexpr and *sexpr′* are the translations of *expr* and *expr′*. This happens no matter what *expr′* is. If *expr′* is just a constant, there is really no need to have a save operation in the target code. If *expr′* is a variable this will be interpreted by first looking up the variable in the environment and then forcing the result of this. This means that the target code for the application would look something like:

```
(sexpr (my-delay (my-force x)))
```

Since the variable `x` is always bound to a delayed value we can replace `(my-delay (my-force x))` by `x`. Similar optimizations can be performed for other kinds of arguments. In the final version of the interpreter interpretation of applications is replaced by:

```
((E (EApply->expr1 expr) venv cenv fenv)
 (EL (EApply->expr2 expr) venv cenv fenv))
```

where `EL` is defined by:

```
(define (EL expr venv cenv fenv ctab fo)
  (cond
   ((EVar? expr)
    (let ([var (EVar->var expr)])
      (if (member var fo)
          (lambda () (venv (EVar->var expr)))
          (venv (EVar->var expr)))))
   ((or (EApply? expr) (EAS? expr) (ELC? expr)
        (E:? expr) (ETuple? expr))
    (my-delay (E expr venv cenv fenv ctab fo idc)))
   (else
    (lambda () (E expr venv cenv fenv ctab fo idc)))))
```

This shows how the optimization is programmed and how the different kinds of arguments are handled.

14.2.2 Restriction of Environments

This section describes a local optimization that may restrict the number of formals in target functions. This was discussed in Section 8.2.2. There are three different functions that are used to restrict environments. The function `restrict-venv` restrict variable environments, and is used several places in the interpreter. The function `restrict-fenv` restrict function environments, and is used before interpreting a single definition. The function `restrict-venv-and-suspend` restrict variable environments while suspending variables bound to non-suspended values. It is used when interpreting local definitions, because we restrict the scope, in which the value of a forced variable can be accessed, to the list of alternatives in an equation. That is, only the suspended value of global variables can be accessed in a local definition. All of the functions take a list of variable names and an environment as input and restrict the environment to names in the list. Besides this `restrict-venv-and-suspend` also takes a list of names of variables that have been forced. As an example, `restrict-venv` is defined by:

```
(define (restrict-venv fv venv)
  (if (null? fv)
      (compile-time-init-venv)
      (let ([n (car fv)])
        (upd n (venv n) (restrict-venv (cdr fv) venv)))))
```

14.2.3 Live Variable analysis

To be able to do the optimization we need functions that returns the free variables of a given syntactical object. It turns out that we need functions that does this for expressions, alternatives and definitions; and a function that returns the function names defined in definitions. These functions can all be written as primitive operations, that is defined in an adt-file, since they are always called with static input. Figure 41 shows a simple free variables analysis for BAWL-programs.

The implementation of FV and DV more or less follows the rules of figure 41. The union $\cup$ and set difference $\setminus$ are defined as higher order functions (of course written in Scheme, but we use a more readable pseudo BAWL notation here):

$f_1 \cup f_2 = \lambda vs.\textbf{let}\ vs_1 = f_2\ vs\ \textbf{in}\ vs_1 +\!+ f_1\ (vs_1 +\!+ vs)$
$f_1 \setminus f_2 = \lambda vs.\ f_1\ (f_2\ vs +\!+ vs)$

and the prefix $\bigcup$ is defined by:

$\bigcup f = \lambda vs.\text{foldr}\ (\cup)\ \text{empty}\ (\text{map f vs})$
where
empty vs = []

to make the implementation more efficient. This means that FV applied to say a list of alternatives is a function that take a list of names as input and returns the free variables in the alternatives except those in the list of names. We can therefore define a function that returns the free variables of a list of alternatives like:

$\text{FV}_{alt*}[\![\, alt*]\!] = \text{FV}[\![\, alt*]\!]\ []$

The code corresponding to FV, DV and the function to compute free variables can all be found in Appendix E.

Free variables:

Scripts and local definitions:

$\mathrm{FV}[\![\, def* \,]\!] = \bigcup_{def \in def*} \mathrm{FV}[\![\, def \,]\!]$

Definitions:

$\mathrm{FV}[\![\, fun\ eq* \,]\!] = \bigcup_{eq \in eq*} \mathrm{FV}[\![\, eq \,]\!]$

Equations:

$\mathrm{FV}[\![\, pat*\ alt*\ \textbf{where}\ def* \,]\!] = (\mathrm{FV}[\![\, alt* \,]\!] \cup \mathrm{FV}[\![\, def* \,]\!]) \setminus \mathrm{DV}[\![\, pat* \,]\!]$

Alternatives:

$\mathrm{FV}[\![\, alt* \,]\!] = \bigcup_{alt \in alt*} \mathrm{FV}[\![\, alt \,]\!]$

$\mathrm{FV}[\![\, exp,\ \textbf{if}\ guard \,]\!] = \mathrm{FV}[\![\, exp \,]\!] \cup \mathrm{FV}[\![\, guard \,]\!]$

Expressions:

$\mathrm{FV}[\![\, var \,]\!] = \{\ var\ \}$

$\mathrm{FV}[\![\, \texttt{(}expr_1\texttt{:}expr_2\texttt{)} \,]\!] = \mathrm{FV}[\![\, expr_1 \,]\!] \cup \mathrm{FV}[\![\, expr_2 \,]\!]$

$\mathrm{FV}[\![\, \texttt{(}expr_1\texttt{,}\ldots\texttt{,}expr_n\texttt{)} \,]\!] = \bigcup_{i=1..n} \mathrm{FV}[\![\, expr_i \,]\!]$

$\mathrm{FV}[\![\, \texttt{(}expr_1\ expr_2\texttt{)} \,]\!] = \mathrm{FV}[\![\, expr_1 \,]\!] \cup \mathrm{FV}[\![\, expr_2 \,]\!]$

$\mathrm{FV}[\![\, expr \,]\!] = \emptyset$, **otherwise**

Arithmetic sequences:

$\mathrm{FV}[\![\, \texttt{[}expr_1\ \texttt{[,}expr_2\texttt{]}\ \texttt{..}\ \texttt{[}expr_3\texttt{]}\ \texttt{]} \,]\!] = \bigcup_{i=1..3} \mathrm{FV}[\![\, expr_i \,]\!]$

List comprehensions:

$\mathrm{FV}[\![\, \texttt{[}expr\texttt{|}pat\texttt{<-}expr_1\texttt{;}qual*\texttt{]} \,]\!] = \mathrm{FV}[\![\, expr_1 \,]\!] \cup (\mathrm{FV}[\![\, \texttt{[}expr\texttt{|}qual*\texttt{]} \,]\!] \setminus \mathrm{DV}[\![\, pat \,]\!])$

$\mathrm{FV}[\![\, \texttt{[}expr\texttt{|}\ expr_1\texttt{;}qual*\texttt{]} \,]\!] = \mathrm{FV}[\![\, expr_1 \,]\!] \cup \mathrm{FV}[\![\, \texttt{[}expr\texttt{|}\ qual*\texttt{]} \,]\!]$

$\mathrm{FV}[\![\, \texttt{[}expr\texttt{|}\ \texttt{]} \,]\!] = \emptyset$

Defined variables:

Definitions:

$\mathrm{DV}[\![\, def* \,]\!] = \bigcup_{def \in def*} \mathrm{DV}[\![\, def \,]\!]$

$\mathrm{DV}[\![\, fun\ eq* \,]\!] = \{\ fun\ \}$

Patterns:

$\mathrm{DV}[\![\, pat* \,]\!] = \bigcup_{pat \in pat*} \mathrm{DV}[\![\, pat \,]\!]$

$\mathrm{DV}[\![\, pat \,]\!] =$ omitted

Figure 41: Free variables

Chapter 15

An Example

In this chapter we give a small example of what a target program generated by the compiler looks like. The target code is shown exactly as produced by the compiler. The example is the following script:

```
fac 0 = 1
fac n = n*fac(n-1)

primes = sieve [2..]
         where
           sieve (p:x) = p:sieve[n|n<-x;n mod p~=0]
```

That is the factorial function `fac` and the Eratosthenes Sieve program to find all primes. The complete target program can be seen in Figure 42. In this target program (or target script) the main function is `S-0`. When `S-0` is called with zero arguments it returns us the information needed to run the compiled script. This information comes in the form of a vector that contains: a constructor environment, a function environment (also containing the conformals), a list of the functions in the script. The structure of these environments are quite simple, so we will not discuss these here, but only take a look at the code corresponding to the definitions in the script.

Let us first look at the compiled function corresponding to `fac`. This is `(fcl-f-0-4 sc_0)`. It can be seen to have a quite natural structure close to the way one would normally write the recursive factorial function in Scheme:

```
(define fac
  (lambda (v)
    (if (= v 0)
        1
        (* v (fac (- v 1))))))
```

The reason that `(fcl-f-0-4 sc_0)` depends on the variable `sc_0` is that the compiler does not detect that `fac` does not depend on `primes`. The predicate `struct-equal?` is being used in stead of just `=` because we are not using type information. The rest of the differences stem from the fact that the language is lazy and we are not using strictness optimization.

The conformal `primes` is not translated into a separate function, since it is not called from within the script. The code:

```
(loadt (string-append **similix-library** "scheme.adt"))
(loadt "bawl.adt")

(define (s-0)
  (let ([cfst_0 (make-cfst 1)])
    (vector
     (lambda (name_1) (standard-constructor name_1))
     (lambda (name_2)
       (cond
        [(equal? name_2 'fac) (fcl-f-0-4 cfst_0)]
        [(equal? name_2 'primes)
         (save-at
          (conf-store-ref cfst_0 0)
          (lambda ()
            ((fcl-f-0-11) (save (lambda () (from 2))))))]
        [else (library-call name_2)]))
     '(fac primes))))

(define (fcl-f-0-11)
  (lambda (v_0)
    (let ([v_1 (v_0)])
      (if (v:? v_1)
          (let* ([v_2 (car v_1)] [v_3 (cdr v_1)])
            (inv: v_2
                  (save (lambda ()
                          ((fcl-f-0-11)
                           (save (lambda () (g-0-16 (v_3) v_2))))))))
          (error '*** "No matching equations for function: ~s" 'sieve)))))

(define (g-0-16 v_0 venv_1)
  (if (vnil? v_0)
      (invnil)
      (let ([v_3 ((car v_0))])
        (if (not-equal? (modulo v_3 (venv_1)) 0)
            (inv: (lambda () v_3)
                  (save (lambda () (g-0-16 ((cdr v_0)) venv_1))))
            (g-0-16 ((cdr v_0)) venv_1)))))

(define (fcl-f-0-4 sc_0)
  (lambda (v_1)
    (let ([v_2 (v_1)])
      (if (equal? 0 v_2)
          1
          (* v_2 ((fcl-f-0-4 sc_0) (save (lambda () (- v_2 1)))))))))
```

Figure 42: Example of a compiled script

```
(save-at
  (conf-store-ref cfst_2 0)
    (lambda ()
      ((fcl-0-10) (save (lambda () (iterate+1 2))))))]
```

above corresponds to `primes`. The `sieve` function on the other hand is called from two places

within the script and there is therefore a function in the target program corresponding to this. This is `(fcl-f-0-11)`. Even the structure of the `sieve` function is fairly straightforward. The reason for the test `v:?` is that the compiler cannot know that `sieve` will never be applied to an empty list. The function `g-0-16` corresponds to the list comprehension in `sieve` and the two parameters of `g-0-16` corresponds the two free variables `x` and `p`.

The quality of the target code produced for the examples is quite satisfactory. The code is small and looks natural except for variable names and a few other details. In the next chapter we will look at the efficiency of target programs and also for the ones in our example.

Chapter 16

Performance

In this chapter we will illustrate the performance of the compilers. We will do so in several ways. Firstly, we will show the speedups gained by partial evaluation and thus by compiling. We will in the same breath show how much is gained be all the optimizations, since we presents run-times and speedups for both compilers. Secondly, we will compare our final compiler to two commercial quality compilers for similar languages: Miranda version 2 from Research Software Ltd. and the LML compiler of Chalmers University. Thirdly, we will compare the performance of our two compilers with respect to compile times, target codes size, etc. Finally, we will shortly compare the results of our efforts with those of "other" compiler generation systems.

Though the intention of this chapter is to give an idea of the state of the compilers and not to document the correctness of these, the rather large and representative set of examples (See Appendix F) that we use in the performance test may to some extend also serves as a simple test of the compiler.

16.1 Strategy for Performance Test

Before presenting the examples and the result of the performance test we have to decide on a strategy for this test.

16.1.1 How to Measure the Speed-up Gained by Partial Evaluation

Usually the speed-up gained by partial evaluation is the ratio between the run-time of the original program and the run-time of the residual program. If the program is an interpreter then this ratio is:

$$\text{speed-up} = \frac{\text{time to interpret program}}{\text{run-time of target program}}$$

but as we shall see this definition is not entirely fair. The problem is that in general one can get arbitrarily large speedups by partial evaluation simply by inserting some big static computation in the program to be specialized. One could of course abstain from using methods like this to get high speedups, but there are still realistic problems where we keep extending the amount of static computation in order to improve the result of the specialization. In cases like this we have to find different strategies for a fair performance test.

Let us look at the problem of specializing an interpreter as we do in this report. If we change the interpreter as described in the previous chapters, then the interpreter will run slower while the target programs will run faster. That is, we get speed-ups for two reasons. On the other hand we may expect the compilers generated from the improved interpreters will run slower, since they have to do the extra static work.

So we believe that a fair strategy would be to compare the time to run a program on the fastest interpreter (i.e. the one not biased towards partial evaluation) against running the target program produced by the best compiler (generated from the final improved version of the interpreter).

Besides this it will be interesting to compare compilers generated from different versions of the interpreters, at least the first and the last ones. By this we mean the compile times and the run-times of the target programs.

16.1.2 How to Compare with Other Systems

To compare our compiler with other compilers for similar languages we again need to find some strategy. A given system might be fast for some types of computation (i.e. list manipulation), but slow for others. This means that first of all we should only compare our system with related system (i.e. for lazy functional languages). Then we should find a set of representative programs and run a test on these. These tests should then hopefully display on which points our system performs better than the systems we compare with and on which points our system performs worse.

16.1.3 How to Measure Run-time and Storage Usage

All the tests were run on a Sparc station 2/Sun OS 4.1. The Scheme system used was Chez Scheme Version 3.2 and the run times in Scheme were measured using the `time` function. The Miranda system used was Miranda version 2 system from Research Software Ltd. and the run times were measured using the `count` option. The version of LML was .99.3 and the run times were measured using the -s option. None of the run times include garbage collection.

We have decided to measure run-times and storage use using the built-in facilities that the three systems supply for this. This is problematic in at least one sense, and that is: we have to rely on the correctness and precision of these facilities. Further more, it looks like Schemes `time` function also include system time in the cpu time while the Miranda `/count` option does not. This means that our time measurements favourize the Miranda system, since the system time is in general around 20 to 30 % of the user time. But still this method will give a good picture of in what the order of magnitude the differences are.

16.2 Run-times and Speed-up

In this section we will show the results of the speed-up test. All run-times are average values of a number of runs in order to make the results reliable. Figure 43 shows a table of the run-times in seconds for two examples. The examples are the two function in the script presented in Chapter 15. The run-times listed under I_1 is for the first and fast interpreter, while those listed under I_2 is for the final version of the interpreter. The run-times listed under T_1 and T_2 are for the target programs produced by the two corresponding compilers.

	I_1	I_2	T_1	T_2
`hd (drop 99 primes)`	5.84 sec	9.59 sec	1.57 sec	0.35 sec
`fac 10`	21 ms	50.6 ms	4.4 ms	1.41 ms

Figure 43: Run-times

Figure 44 shows speed-ups. The ratios listed under I_1/T_2 is the speed-up gained by compiling, that is how much faster the target program of the final compiler runs faster than the same program interpreted by the first interpreter. The ratios listed under I_1/T_1 and I_2/T_2 shows the

actual speedup by partial evaluation of the interpreters and the ratios listed under T_1/T_2 show how much is gained by the rewriting of the interpreter.

	I_1/T_1	I_2/T_2	I_1/T_2	T_1/T_2
`hd (drop 99 primes)`	3.7	27.4	16.7	4.5
`fac 10`	4.8	35.9	14.9	3.1

Figure 44: Speed-ups

These results corresponds very much to what we would expect from the discussion above. Especially we can see that the speed-up that we gain from rewriting the interpreter is around 4 which is quite satisfactory.

16.3 The Performance Compared with the Miranda and LML

In this section we will compare our final compiler's performance with two compilers for similar languages. The systems we compare with is Miranda version 2 from Research Software and the LML version .99.3. The selection of examples used can be found in Appendix F. This selection of examples all use functions that are given as examples in the Miranda manual [Turner 1989], though some have been modified slightly to stay within the BAWL language (i.e. plus patterns have been removed). The LML test programs are as close in structure to the corresponding Miranda/BAWL programs as possible. The cases missing for LML (marked with -) could either not be run correctly or were hard to write in a way close to the corresponding BAWL-program. The examples are constructed such that they have about the same run-time (not so easy in practice). Figure 45 shows the result of the test. Times are in second and the storage allocated are in kbytes.

16.4 The Compilers

In this section we will give some statistics about the compilers. The compilers were generated by using Similix's cogen function. Figure 46 shows the statistics for the two compilers. Com1 is the first and simple compiler and Com2 is the final one. Time_{pre} is the time to preprocess the interpreters, Time_{gen} is the time to compute the compilers when the interpreters have been preprocessed (i.e. cogen(Int)), No. fun. is the number of functions in the compiler, Size_{comp} is the size of the compiler in bytes (measured by use of wc in UNIX). Time_{comp} the time used to compile the entire collections of test functions of the last section (found in Appendix F and Size_{target} is the size of the target program.

A surprising result is that the final compiler compiles our test program faster than the first compiler. One explanation of this could be that the final compiler has to produce less code. This then results in less call to primitive that generate code, a faster check to see if a residual function has been generated before and less time to do postprocessing. A simple test, that we have performed, showed that the time used on postprocessing the code by Similix dominates the total runtime at least in the example above.

16.5 Sharing of Code

In section 13 we described how to solve what we named: the code duplication problem. The result was to get more sharing of code in the target programs. The final compiler presented in the report does use this optimization. Compiling the example file of the last section without sharing

	Run-times/sec			Storage allocated/kbytes		
	Miranda	BAWL	LML	Miranda	BAWL	LML
gcdtest	0.57	0.20	0.033	511	327	30
quadsolvetest	0.42	0.35	0.085	407	406	175
fact0est	2.33	0.74	1.75	2464	1461	2020
fac1test	2.98	1.07	4.17	3328	2126	9025
fac2test	2.82	2.13	4.36	3329	5249	9268
acktest	1.22	0.42	0.061	1485	1587	165
fib0test	2.35	0.48	0.08	2054	1227	449
fib1test	0.33	0.76	0.45	86	79	857
fib2test	0.32	0.76	0.44	53	76	854
answertest	2.58	2.36	-	2065	6294	-
reverse1test	0.28	0.56	0.094	408	1095	264
reverse2test	0.32	0.58	0.12	430	1137	264
permstest	1.17	0.41	-	1501	853	-
factortest	0.80	0.88	0.28	776	822	133
qsorttest1	0.37	0.26	0.10	306	393	92
qsorttest2	0.48	0.40	0.08	395	651	95
queenstest	0.97	0.61	0.18	964	1146	497
perfectstest	4.43	4.42	1.59	4262	4251	593
primestest	0.42	0.35	0.07	379	409	102
hammingtest	0.72	0.32	0.096	979	689	126
treetest	1.38	0.64	0.09	1107	1103	356

Figure 45: Compared performance

	Com1	Com2
Time_{pre}/sec.	5.32	37.3
Time_{gen}/sec.	12.6	68.3
No. fun.	66	228
Size_{comp}/kbytes	76.8	233
Size_{comp}/cells	11425	35635
Time_{comp}/sec.	14.9	7.60
Size_{target}/kbytes	38.6	21.0
Size_{target}/cells	8691	4460

Figure 46: Compared performance of the compilers

of code gives a slightly bigger program (size_{target}/cells = 4779). But many of the functions in the target program with sharing has a lot of superfluous parameter that could be removed by a simple optimization (see chapter 17) that could in general make these programs even smaller than the ones without sharing of code.

16.6 Comparison with Other compiler generation systems

An exact or even a fair comparison of compiler generation system will be virtually impossible. The compiler generator systems in this section have been build for different purposes, needing a different amount of manual assistance and runs on different hard ware. All this makes it very hard to compare compile and run times of the systems. We will present a few results anyway to give an idea of what is possible to achieve by complier generator systems and especially what may be possible to achieve by complier generators based on partial evaluation.

The first system we compare with is CERES (Jones, Christiansen [Christiansen and Jones 1982], Tofte). CERES used 3 sec. to compile a factorial program and 0.30 sec. to calculate factorial of 20. The tests were run on a VAX 11/750.

For the system QUOKKA by Trevor Vickers [Vickers 1986] no compiler generation times are reported, but the cpu-time used to calculate factorial of 10 was 0.28 sec. and the size of the target program was 5240 bytes. Again this test was run on a VAX 11/750.

The last system we give results for is SAM by Pierre Weis [Weis 1987]. SAM used 0.22 sec. to compile a factorial program and 0.26 sec. cpu-time to calculate factorial of 100. SAM runs on a VAX 11/780.

Compared to these systems our compiler compiles the factorial program of section 15 in 50 ms. (not including parsing and type checking) and calculates factorial of 10 in 1.4 ms, factorial of 20 in 2.0 ms. and factorial of 100 in 8 ms. The size of the target program is 531 bytes. To be fair our tests were run on a Sparc station 2.

16.7 Conclusion on the Test

Our test shows that it is possible to generate a compile that produces reasonable code both regarding size and speed. We can also conclude that the improvement of the original interpreter did result in a compiler generating better code and surprisingly: faster than the simple non optimizing compiler.

We can in general state that our implementation is as fast or maybe a little faster than the Miranda system and and uses about the same amount of storage. We can on the other hand not compete with the graph reduction based LML system, neither with respect to speed nor use of storage.

Chapter 17

Discussion

This chapter concludes the second and last part of this thesis. We will do so by first discussing some natural extensions and improvements to the current version of our compiler. Then we will discuss future work in general and finally conclude on the results obtained by the case study part of this project.

17.1 Further improvements of the compiler

There is still room for improvements of the compiler, both by further rewriting of the interpreter or by transformations of either the source programs or the target programs.

As described in section 12.3.2 all target functions get extra parameters to allow them to access the values of conformals, and this happens even if the functions never actually need to access any of these values. A simple analysis of the target programs may detect these superfluous parameters and eliminate them.

A related problem is superfluous local definitions. Target programs may have let-expressions in them binding variables that are never referenced in the body. The reason for this is that Similix only unfolds let-expressions when the the let-variable is used exactly once. This prevents both duplication and discarding of expressions. In the case of discarding the problem is that discarding expressions may remove non-termination thus changing the semantics of the program in question. A simple termination analysis may remove a lot of these bindings.

In the Scheme code generated by the compiler many functions are curried, because they inherit their structure from the source language. Uncurrying some of these functions could improve performance considerably (simple examples have shown this, see also [Gomard 1991]) because tupled application is much faster in Scheme (fewer closures have to be built). Which functions to uncurry could be decided by a simple analysis of the source program and the result of this could be used by the interpreter, and this would then make these target functions appear in uncurried form. This analysis may also be performed on the source language and the optimizations be inserted in the interpreter.

Strictness analysis of the source program could also be used to improve the target programs. Using strictness information would in the case of the factorial function give a speedup of the generated code of about 3 times.

Preliminary experiments have shown that also lambda lifting of the source programs may yield good results, but this is a field that needs more investigation. A related, but simpler optimization could be called conformal lifting. Consider the following BAWL version of the fibonacci function:

```
fib n = 1,                          if n<2
      = flist!(n-1) + flist!(n-2), otherwise
        where
          flist = map fib [0..]
```

This program is excessively slow, since `flist` is evaluated each time fib is called, because our compiler "thinks" that `flist` can depend on `n`. However, `flist` does not depend on `n` and we could safely move the definition of `flist` out to the level of `fib`. If we do this we get a program that is the original program quite superior in speed and space consumption.

If a type-checker were written, the result of the type-checking could be used to optimize the target code, e.g. replace polymorphic equality by specific instances.

Since Scheme is dynamically typed, Scheme implementations spend a lot of time doing type checking when running programs and this time is wasted if we know that a program is "well typed." Since the target programs produced by our compiler are all well typed, it should be possible to run these without the dynamic type checking and thereby improve further on their performance. One solution could be to generate code for a different strict, but typed language like for instance Standard ML [Milner *et al.* 1990] and this might be possible by rewriting the part of Similix that generates code. Another solution would be to have a Scheme compiler in which one could turn off the type checking.

17.2 Future Work

Future work could be to implement some of the improvements discussed above. It would also be interesting to see if one could change the interpreter to obtain some form of separate compilation of module and a method to link modules together. One might also try to modify the interpreter to compile a known language, for example Haskell. Instrumenting the interpreter with debug or trace facilities would also be an interesting project.

17.3 Conclusion

We have shown that it is possible by partial evaluation to generate compilers for languages of a realistic size and that the compilers generated can in fact produce reasonably fast target code even compared to handwritten compilers. This is in fact a surprising result, since partial evaluation is a very general principle and efficient application of principle are often not successful. One may argue that we used a lot of clever "tricks" to obtain this good result, but as our performance test showed we still got quite good results without the "tricks". On top of this, it seems to be possible to automate some of these "tricks".

Appendix A

Additional Proofs

A.1 Correctness Proof of the Type Translation

We will in this appendix formalize what it means for our type transformation $\mathbf{T}$ to be correct and prove that it is so. We will thereby show that: if e_σ is well typed with type σ, then $\mathbf{T}[\![e_\sigma]\!]$ is well typed with type $\mathbf{T}[\![\sigma]\!]$. This was what we claimed in chapter 5.

First we must formalize the notion that two environments relates.

Definition A.1 We define a relation on type environments $\equiv \in$ Type-Env $\times$ Type-Env (Type-Env = Var $\rightarrow$ Type), by:

$$\Gamma_1 \equiv \Gamma_2 \textbf{ iff } \mathrm{Dom}(\Gamma_1) = \mathrm{Dom}(\Gamma_2) \wedge \forall\, \mathrm{x} \in \mathrm{Dom}(\Gamma_1)\colon\ \Gamma_1(\mathrm{x}) = \mathrm{Unit} \rightarrow \mathbf{T}[\![\Gamma_1(\mathrm{x})]\!]$$

Proposition A.1 If $\Gamma_L \equiv \Gamma_S$ then: for all expressions e_σ it hold that:

$$\Gamma_L \vdash e_\sigma\colon \sigma \Rightarrow \Gamma_S \vdash \mathbf{T}[\![e_\sigma]\!]\colon \mathbf{T}[\![\sigma]\!]$$

Proof: The proof is by structural induction over the structure of e_σ.
Case: x.

$$\begin{aligned}\Gamma_L \vdash x_\sigma\colon \sigma &\Rightarrow \Gamma_L(x) = \sigma \\ &\Rightarrow \Gamma_S(x) = \mathrm{Unit} \rightarrow \mathbf{T}[\![\sigma]\!] \\ &\Rightarrow \Gamma_S \vdash x \texttt{ \#}\colon \mathbf{T}[\![\sigma]\!] \\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![x_\sigma]\!]\colon \mathbf{T}[\![\sigma]\!]\end{aligned}$$

Case: #.

Trivial.

Case: $\lambda x_\tau.e_\sigma$. Let $\tau' = \mathbf{T}[\![\tau]\!]$ and $\sigma' = \mathbf{T}[\![\sigma]\!]$, then

$$\begin{aligned}\Gamma_L \vdash \lambda x_\tau.e_\sigma\colon \tau \rightarrow \sigma &\Rightarrow [x \mapsto \tau]\Gamma_L \vdash e_\sigma\colon \sigma \\ &\Rightarrow [x \mapsto (\mathrm{Unit} \rightarrow \tau')]\Gamma_S \vdash \mathbf{T}[\![e_\sigma]\!]\colon \sigma' \\ &\Rightarrow \Gamma_S \vdash \lambda x_{Unit \rightarrow \tau'}.\mathbf{T}[\![e_\sigma]\!]\colon (\mathrm{Unit} \rightarrow \tau') \rightarrow \sigma' \\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![\lambda x_\tau.e_\sigma]\!]\colon \mathbf{T}[\![\tau \rightarrow \sigma]\!]\end{aligned}$$

Case: $e_{\tau \rightarrow \sigma}\ e'_\tau$.

$$\begin{aligned}\Gamma_L \vdash e_{\tau\to\sigma}\ e'_\tau\colon \sigma &\Rightarrow \Gamma_L \vdash e_{\tau\to\sigma}\colon \tau\to\sigma \wedge \Gamma_L \vdash e'_\tau\colon \tau\\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![e_{\tau\to\sigma}]\!]\colon (\text{Unit}\to\mathbf{T}[\![\tau]\!])\to\mathbf{T}[\![\sigma]\!] \wedge\\ &\quad\ \Gamma_S \vdash \mathbf{T}[\![e'_\tau]\!]\colon \mathbf{T}[\![\tau]\!]\\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![e_{\tau\to\sigma}]\!]\colon (\text{Unit}\to\mathbf{T}[\![\tau]\!])\to\mathbf{T}[\![\sigma]\!] \wedge\\ &\quad\ \Gamma_S \vdash \lambda\alpha_{Unit}.\mathbf{T}[\![e'_\tau]\!]\colon \text{Unit}\to\mathbf{T}[\![\tau]\!]\\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![e_{\tau\to\sigma}\ e'_\tau]\!]\colon \mathbf{T}[\![\sigma]\!]\end{aligned}$$

Case: `rec` $x_\tau\ e_\tau$. Let $\tau' = \mathbf{T}[\![\tau]\!]$, then

$$\begin{aligned}\Gamma_L \vdash \texttt{rec}\ x_\tau\ e_\tau\colon \tau &\Rightarrow [x\mapsto\tau]\Gamma_L \vdash e_\tau\colon \tau\\ &\Rightarrow [x\mapsto(\text{Unit}\to\tau')]\Gamma_S \vdash \mathbf{T}[\![e_\tau]\!]\colon \tau'\\ &\Rightarrow [x\mapsto\tau']\Gamma_S \vdash \lambda x_{Unit\to\tau'}.\mathbf{T}[\![e_\tau]\!]\colon (\text{Unit}\to\tau')\to\tau' \wedge\\ &\quad\ [x\mapsto\tau']\Gamma_S \vdash \lambda\alpha_{Unit}.x_{\tau'}\colon \text{Unit}\to\tau'\\ &\Rightarrow [x\mapsto\tau']\Gamma_S \vdash (\lambda x_{Unit\to\tau'}.\mathbf{T}[\![e_\tau]\!])\ (\lambda\alpha_{Unit}.x_{\tau'})\colon \tau'\\ &\Rightarrow \Gamma_S \vdash \mathbf{T}[\![\texttt{rec}\ x_\tau\ e_\tau]\!]\colon \mathbf{T}[\![\tau]\!]\end{aligned}$$

which complete the proof. □

Corollary A.1 If e_σ is well typed with type σ, then $\mathbf{T}[\![e_\sigma]\!]$ is well typed with type $\mathbf{T}[\![\sigma]\!]$.

□

A.2 Proof of the Applicative Fixed Point Operator

In this section we give a formal proof of the correctness of the applicative fixed point operator. We will prove this by proving a proposition about lambda calculus expressions, but we will first motivate why this is indeed what we are interested in. Assume that we have a lambda calculus based strict functional language[1] (e.g. Scheme) and we have the following definitions:

```
fix = λf.λx.f (fix f) x
f   = fix (λg.λx.expr)
```

where *expr* is some arbitrary expressions in the language. Then the meaning that we assign to `f` is of the form:

$$\mathbf{fix}(\lambda\text{fix}.\underline{\lambda}\text{f}.\underline{\lambda}\text{x.f (fix f) x})(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)$$

where e is the meaning that we assign to *expr*. If the function `fix` behaves like the fixed point operator for functionals defining functions, then the meaning of `f` should be the same as the meaning of the function `g`, defined by:

```
g = λx.expr
```

The meaning assigned to `g` has the form:

$$\mathbf{fix}(\lambda\text{g}.\underline{\lambda}\text{x}.e)$$

and we are therefore interested in proving the following proposition:

[1] In a strict language, like Scheme, we have $\lambda\text{x}.\bot \neq \bot$ in the value domain.

Proposition A.2 If e is a lambda calculus expression, then:

$$\mathbf{fix}(\lambda\text{fix}.\underline{\lambda}\text{f}.\underline{\lambda}\text{x.f (fix f) x})(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) = \mathbf{fix}(\lambda\text{g}.\underline{\lambda}\text{x}.e)$$

Proof: The proof is by fixed point induction. Let

$$\begin{aligned}\text{F} &= \lambda\text{h}.\underline{\lambda}\text{f}.\underline{\lambda}\text{x.f (h f) x}\\ \text{G} &= \lambda\text{h}.\underline{\lambda}\text{x}.e\end{aligned}$$

We will use the notation $e[e']$ to mean $e[e'/\text{g}]$ (i.e. e in which e' is substituted for g). We can now reformulate the statement that we want to prove as:

$$(\bigsqcup_{\text{n}=0}^{\infty} \text{F}^{\text{n}}(\bot))(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) = \bigsqcup_{\text{n}=0}^{\infty} \text{G}^{\text{n}}(\bot)$$

which is equivalent to:

$$\bigsqcup_{\text{n}=0}^{\infty} (\text{F}^{\text{n}}(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)) = \bigsqcup_{\text{n}=0}^{\infty} \text{G}^{\text{n}}(\bot)$$

Unfortunately we cannot prove equality of each corresponding pairs of elements in the two chains, but we can prove the following which is just as good:

$$\text{G}^{\text{n}}(\bot) \sqsubseteq \text{F}^{\text{n}+1}(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) \sqsubseteq \text{G}^{\text{n}+1}(\bot) \tag{A.1}$$

for all $\text{n} \geq 0$, which then is our induction hypothesis.

Case: Base (n = 0).

$$\text{G}^0(\bot) \sqsubseteq \text{F}^1(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)) \sqsubseteq \text{G}^1(\bot) \Rightarrow \bot \sqsubseteq \underline{\lambda}\text{x}.\bot \sqsubseteq \underline{\lambda}\text{x}.e[\bot]$$

Case: Inductive ($\text{n} \geq 1$).

Assume that we have:

$$\text{G}^{\text{n}-1}(\bot) \sqsubseteq \text{F}^{\text{n}}(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) \sqsubseteq \text{G}^{\text{n}}(\bot)$$

We start by a small rewriting:

$$\begin{aligned}\text{F}^{n+1}(\bot)\ (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) &= (\underline{\lambda}\text{f}.\underline{\lambda}\text{x.f}\ (\text{F}^n(\bot)\ \text{f})\ \text{x})\ (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)\\ &= \underline{\lambda}\text{x}.(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)\ (\text{F}^n(\bot)\ (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e))\ \text{x}\\ &= (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)\ (\text{F}^n(\bot)\ (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e))\end{aligned}$$

The last equality is proved by assuming the two sides applied to $\bot$ and to a value different from $\bot$, and showing the results correct in both cases. Thus, by the monotonicity of $(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)$ and the induction hypothesis we have:

$$\text{F}^{\text{n}+1}(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) \sqsupseteq (\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e)\text{G}^{\text{n}-1}(\bot) = \text{G}^{\text{n}}(\bot)$$

for $\text{n} > 1$ because then we have $\text{G}^n(\bot) \neq \bot$. For n = 1 we have to check the equality separately. Via a few rewriting we get

$$\text{F}^2(\bot)(\underline{\lambda}\text{g}.\underline{\lambda}\text{x}.e) = \underline{\lambda}\text{x}.e[\underline{\lambda}\text{x}.\bot] \sqsupseteq \underline{\lambda}\text{x}.e[\bot] = \text{G}^1(\bot)$$

We have therefore shown that

$$\mathrm{G}^{\mathrm{n}}(\bot) \sqsubseteq \ \mathrm{F}^{\mathrm{n}+1}(\bot)(\underline{\lambda}\mathrm{g}.\underline{\lambda}\mathrm{x}.e)$$

holds for $\mathrm{n} \geq 1$, which was half of what we had to show. Now we turn to the other half which is quite easy to show. Again by the monotonicity of $(\underline{\lambda}\mathrm{g}.\underline{\lambda}\mathrm{x}.e)$ and the induction hypothesis we have:

$$\mathrm{F}^{\mathrm{n}+1}(\bot)(\underline{\lambda}\mathrm{g}.\underline{\lambda}\mathrm{x}.e) \sqsubseteq \ (\underline{\lambda}\mathrm{g}.\underline{\lambda}\mathrm{x}.e)\mathrm{G}^{\mathrm{n}}(\bot) = \mathrm{G}^{\mathrm{n}+1}(\bot)$$

This means that (A.1) holds for all finite n's.

Case: Limit.

Trivial. □

Appendix B

An Extended Type Checking System

In this appendix we show without any proof an extended versionof the the system of Figure 11. The type rules are for an extended language. The syntax of expressions in this language is show in figure 47. The new kind of expressions are a conditional expressions; a pair constructor `(_,_)` and corresponding projections `fst` and `snd`; sum domain injectors `inr` and `inl`; and a cases-of expression.

Abstract syntax:

$e \in \text{Exp}$, $x \in \text{Var}$

$e ::= x \mid c \mid \lambda x.e \mid \underline{\lambda} x.e \mid e\ e \mid \texttt{if}\ e\ e\ e \mid \texttt{(}e\texttt{,}e\texttt{)} \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid$
$\texttt{inr}\ e \mid \texttt{inl}\ e \mid \texttt{cases}\ e\ \texttt{of}\ \texttt{isr?(}x\texttt{):}e\ \ \texttt{isl?(}x\texttt{):}e \mid \texttt{fix}\ \lambda x.e$

$\tau \in \text{Type}$, $\alpha \in \text{IType}$

$\tau ::= \text{v}(\alpha) \mid \text{s}(\alpha)$

$\alpha ::= \text{Base}$ (base types)
$\mid \tau \rightarrow \tau$ (function type)
$\mid \texttt{(}\tau\texttt{,}\tau\texttt{)}$ (product type)
$\mid \texttt{[}\tau\texttt{,}\tau\texttt{]}$ (sum type)

Figure 47: Syntax of the extended language and types

The complete type checking system for the extended language is shown in Figure 48.

Type rules:

(CONST) $\Gamma \vdash c\colon \mathrm{v}(\mathrm{Base})$

(VAR) $$\frac{\Gamma(x) = \tau_x \quad \mathrm{c}\colon \tau_x \geq_{\downarrow} \tau}{\Gamma \vdash \mathrm{c}x\colon \tau}$$

(ABS) $$\frac{[x \mapsto \mathrm{v}(\alpha_x)]\Gamma \vdash e\colon \mathrm{v}(\alpha)}{\Gamma \vdash \underline{\lambda} x.e\colon \mathrm{v}(\mathrm{v}(\alpha_x) \to \mathrm{v}(\alpha))}$$

(ABS) $$\frac{[x \mapsto \mathrm{s}(\alpha_x)]\Gamma \vdash e\colon \mathrm{v}(\alpha)}{\Gamma \vdash \lambda x.e\colon \mathrm{v}(\mathrm{s}(\alpha_x) \to \mathrm{v}(\alpha))}$$

(APP) $$\frac{\Gamma \vdash e\colon \mathrm{v}(\tau \to \mathrm{v}(\alpha)) \quad \Gamma \vdash e'\colon \tau' \quad \mathrm{c}_1\colon \tau' \leq_{\uparrow} \tau'' \quad \mathrm{c}_2\colon \tau'' \leq_{\Uparrow} \tau}{\Gamma \vdash e \;\; \mathrm{c}_1\mathrm{c}_2 e'\colon \mathrm{v}(\alpha)}$$

(IF) $$\frac{\Gamma \vdash e_t\colon \mathrm{v}(\mathrm{Base}) \quad \Gamma \vdash e_c\colon \tau_c \quad \Gamma \vdash e_a\colon \tau_a \quad \mathrm{c}_c\colon \tau_c \leq_{\Uparrow} \mathrm{v}(\alpha) \quad \mathrm{c}_a\colon \tau_a \leq_{\Uparrow} \mathrm{v}(\alpha)}{\Gamma \vdash \texttt{if}\ e_t \;\; \mathrm{c}_c e_c \;\; \mathrm{c}_a e_a\colon \mathrm{v}(\alpha)}$$

(FIX) $$\frac{[x \mapsto \mathrm{v}(\alpha)]\Gamma \vdash e\colon \mathrm{v}(\alpha)}{\Gamma \vdash \texttt{fix}\ \lambda x.e\colon \mathrm{v}(\alpha)}$$

(PAIR) $$\frac{\Gamma \vdash e_r\colon \tau_r \quad \Gamma \vdash e_l\colon \tau_l \quad \mathrm{c}_r\colon \tau_r \leq_{\uparrow} \mathrm{s}(\alpha_r) \quad \mathrm{c}_l\colon \tau_l \leq_{\uparrow} \mathrm{s}(\alpha_l)}{\Gamma \vdash (\mathrm{c}_r e_r, \mathrm{c}_l e_l)\colon (\mathrm{s}(\alpha_r), \mathrm{s}(\alpha_l))}$$

(FST) $$\frac{\Gamma \vdash e\colon \mathrm{v}((\mathrm{s}(\alpha_r), \mathrm{s}(\alpha_l))) \quad \mathrm{c}\colon \mathrm{s}(\alpha_r) \leq_{\downarrow} \mathrm{v}(\alpha_r)}{\Gamma \vdash \mathrm{c}(\texttt{fst}\ e)\colon \mathrm{v}(\alpha_r)}$$

(SND) (similar to the FST rule)

(INR) $$\frac{\Gamma \vdash e\colon \tau \quad \mathrm{c}\colon \tau \leq_{\uparrow} \mathrm{s}(\alpha_r)}{\Gamma \vdash \texttt{inr}\ \mathrm{c}e\colon \mathrm{v}([\mathrm{s}(\alpha_r), \mathrm{s}(\alpha_l)])}$$

(INL) (similar to the INR rule)

(CASES) $$\frac{\Gamma \vdash e\colon \mathrm{v}([\mathrm{s}(\alpha_r), \mathrm{s}(\alpha_l)]) \quad [x_r \mapsto \mathrm{s}(\alpha_r)]\Gamma \vdash e_r\colon \mathrm{v}(\alpha) \quad [x_l \mapsto \mathrm{s}(\alpha_l)]\Gamma \vdash e_l\colon \mathrm{v}(\alpha)}{\Gamma \vdash \texttt{cases}\ e\ \texttt{of isr?}(x_r)\texttt{:}e_r \;\; \texttt{isl?}(x_l)\texttt{:}e_l\colon \mathrm{v}(\alpha)}$$

Coercion relations:

$\mathrm{nop} : \tau \geq_{\downarrow} \tau$ $\qquad$ $\mathrm{nop} : \tau \leq_{\uparrow} \tau$

$\downarrow \;\; : \mathrm{s}(\alpha) \geq_{\downarrow} \mathrm{v}(\alpha)$ $\qquad$ $\uparrow \;\; : \mathrm{v}(\alpha) \leq_{\uparrow} \mathrm{s}(\alpha)$

$\mathrm{nop} : \tau \leq_{\Uparrow} \tau$

$\Uparrow \;\; : t(\mathrm{v}(\alpha_1) \to \mathrm{v}(\alpha_2)) \leq_{\Uparrow} t(\mathrm{s}(\alpha_1) \to \mathrm{v}(\alpha_2)) \quad t = \mathrm{s}$ **or** $t = \mathrm{v}$

Figure 48: The extended type checking system

Appendix C

The First Interpreter

```
; -*- Scheme -*- mode
;-------------------------------------------------------------------------------
; File : bawl0.sim.
;-------------------------------------------------------------------------------
(load  "bawl0.syn")
(loadt "bawl0.adt")
(loadt "thunk0.adt")
(loadt (string-append **Similix-library** "scheme.adt"))
;-------------------------------------------------------------------------------
; Main function:
;
(define (S scr)
  (let* ([scr  (elab-scr scr)]
         [funs (collect-functions scr)]
         [cenv (CE scr)]
         [fenv (FE scr (init-venv) cenv (init-fenv))])
    (vector
      (make-env-static (collect-constructors scr) cenv (init-cenv))
      (make-env-static funs fenv (init-fenv))
      funs)))
;
;==============================================================================
; Constructor environment:
;
(define (CE def*)
  (if (null? def*)
      (init-cenv)
      (let ([def1 (def*->def1 def*)])
        (if (TypeDef? def1)
            (CD (def->cdef* def1) (CE (def*->def* def*)))
            (CE (def*->def* def*))))))
;
(define (CD cdef* cenv)
  (if (null? cdef*)
      cenv
      (let* ([cdef (cdef*->cdef1 cdef*)]
             [con  (cdef->con cdef)]
             [a    (length (cdef->atexpr* cdef))])
        (upd con (lambda ()
                   (fcl a (lambda (vs)
                            (inVCon con (listToTuple vs)))))
             (CD (cdef*->cdef* cdef*) cenv)))))
;
```

```
;=============================================================================
; Function environment:
;
(define (FE scr venv cenv fenvi)
  (let* ((ncfs (collect-conformals scr))   ; New conformals at this level.
         (cfst (make-cfst (length ncfs))))
    (fix
     (lambda (fenv)
       (lambda (name)
         (FD scr name venv cenv fenv fenvi cfst ncfs))))))
;
(define (FD def* name venv cenv fenv fenvi cfst ncfs)
  (if (null? def*)
      (fenvi name)
      (let ([def (def*->def1 def*)])
        (if (and (FunDef? def) (equal? name (Def->fun def)))
            (let* ([fun (Def->fun def)]
                   [eq* (Def->eq* def)])
              (if (member name ncfs)
                  (save-at
                   (conf-store-ref cfst (location name ncfs))
                   (lambda ()
                     (ME eq* venv cenv fenv '() name)))
                  (fcl (Funarity eq*)
                       (lambda (vs)
                         (ME eq* venv cenv fenv vs name)))))
            (FD (def*->def* def*) name venv cenv fenv fenvi cfst ncfs)))))
;
(define (ME eq* venv cenv fenv vs name)
  (if (null? eq*)
      (error '*** "No matching equations for function: ~s" name)
      (let* ([eq   (eq*->eq1 eq*)]
             [pat* (eq->pat* eq)]
             [alt* (eq->alt* eq)]
             [def* (eq->def* eq)])
        (MP pat* venv
            (lambda (venv1)
              (MA alt* venv1 cenv (FE def* venv1 cenv fenv) name))
            (lambda () (ME (eq*->eq* eq*) venv cenv fenv vs name))
            vs))))
;
(define (MA alt* venv cenv fenv name)
  (if (null? alt*)
      (error '*** "No matching alternatives for function: ~s" name)
      (let* ([alt   (alt*->alt  alt*)]
             [expr  (alt->expr alt)]
             [guard (alt->guard alt)])
        (if (E guard venv cenv fenv)
            (E expr venv cenv fenv)
            (MA (alt*->alt* alt*) venv cenv fenv name)))))
;
```

```
;=============================================================================
; Expressions:
;
(define (E expr venv cenv fenv)
  (cond
   ((ELit?   expr)     (L (ELit->lit expr)))
   ((EVar?   expr)     (venv (EVar->var expr)))
   ((EFun?   expr)     (fenv (EFun->fun expr)))
   ((ECon?   expr)     (cenv (ECon->con expr)))
   ((EOp?    expr)     (B (EOp->op expr)))
   ((EBool?  expr)     (Bool expr))
   ((ENil?   expr)     (inVNil))
   ((E:?     expr)     (inV: (my-delay (E (E:->expr1 expr) venv cenv fenv))
                           (my-delay (E (E:->expr2 expr) venv cenv fenv))))
   ((ETuple? expr)     (listToTuple (E* (ETuple->expr* expr) venv cenv fenv)))
   ((EApply? expr)     ((E (EApply->expr1 expr) venv cenv fenv)
                        (my-delay (E (EApply->expr2 expr) venv cenv fenv))))
   ((EAS?    expr)     (AS expr venv cenv fenv))
   ((ELC?    expr)     (LC expr cenv fenv venv (lambda () (inVNil))))
   (else               (error '*** "Syntax error in expression: ~s" expr))))
;
(define (E* expr* venv cenv fenv)
  (if (null? expr*)
      '()
      (cons (my-delay (E (car expr*) venv cenv fenv))
            (E* (cdr expr*) venv cenv fenv))))
;
(define (L lit)
  (cond
   ((LNum? lit) (Lit->num lit))
   ((LChar? lit) (Lit->char lit))
   (else ; lit a string
    (lit->String lit))))
;
;=============================================================================
; Arithmetic Sequences:
;
(define (AS expr venv cenv fenv)
  (let* ([expr1 (Eas->expr1 expr)]
         [expr2 (Eas->expr2 expr)]
         [expr3 (Eas->expr3 expr)])
    (cond
     ((AS-e..? expr)   ; expr has form [expr..]
      (from (E expr1 venv cenv fenv)))
     ((AS-ee..? expr)  ; expr has form  [expr,expr..]
      (let ([v1 (E expr1 venv cenv fenv)])
        (fromThen v1 (- (E expr2 venv cenv fenv) v1))))
     ((AS-e..e? expr)  ; expr has form [expr..expr]
      (fromTo (E expr1 venv cenv fenv) (E expr3 venv cenv fenv)))
     (else             ; expr has form [expr,expr..expr]
      (let* ([v1 (E expr1 venv cenv fenv)]
             [v3 (E expr3 venv cenv fenv)]
             [s (- (E expr2 venv cenv fenv) v1)])
        (if (>= s 0)
            (fromUpTo v1 s v3)
            (fromDownTo v1 s v3)))))))
;
```

```
;=============================================================================
; List comprehension:
;
(define (LC elc cenv fenv venv vs)
  (let* ([expr0 (ELC->expr elc)] [qual* (ELC->qual* elc)])
    (if (null? qual*)
        (inV: (my-delay (E expr0 venv cenv fenv)) vs)
        (let* ([qual (car qual*)] [qual1* (cdr qual*)])
          (if (LCgen? qual)
              (G (LCgen->pat qual)
                 (lambda (venv1 v1)
                   (LC (inLC expr0 qual1*) cenv fenv venv1 v1))
                 venv vs (E (LCgen->expr qual) venv cenv fenv))
              (if (E (LCfilter->expr qual) venv cenv fenv)
                  (LC (inLC expr0 qual1*) cenv fenv venv vs)
                  (my-force vs)))))))
;
(define (G pat c venv vs1 vs)
  (if (vnil? vs)
      (my-force vs1)
      (P pat venv
         (lambda (venv1)
           (c venv1 (my-delay (G pat c venv vs1 (my-force (cdr vs))))))
         (lambda () (G pat c venv vs1 (my-force (cdr vs))))
         (car vs))))
;
;=============================================================================
; Pattern matching :
;
(define (MP pat* venv sc fc vs)
  (if (null? pat*)
      (sc venv)
      (P (pat*->p1 pat*) venv
         (lambda (venv1) (MP (pat*->pat* pat*) venv1 sc fc (cdr vs)))
         fc (car vs))))
;
(define (P pat venv sc fc v)
  (cond
   ((PVar? pat)
    (sc (upd (Pvar->var pat) v venv)))
   ((Plit? pat)
    (if (vequal? (my-force v) (L (Plit->lit pat))) (sc venv) (fc)))
   ((Pcon? pat)
    (let ([fv (my-force v)] [pat* (Pcon->pat* pat)])
      (if (equal? (VCon->con fv) (Pcon->con pat))
          (MP pat* venv sc fc (tupleToList (length pat*) (lambda () fv)))
          (fc))))
   ((Ppair? pat)
    (let ([fv (my-force v)])
      (if (pair? fv)
          (P (Ppair->p1 pat) venv
             (lambda (venv1) (P (Ppair->p2 pat) venv1 sc fc (cdr fv)))
             fc (car fv))
          (fc))))
   ((Pnil? pat)
    (if (vnil? (my-force v)) (sc venv) (fc)))
   ((PTuple? pat)
    (let ([pat* (PTuple->pat* pat)])
      (MP pat* venv sc fc (tupleToList (length pat*) v))))
   (else
    (error '*** "Syntax error in pattern: ~s" pat))))
;
```

```
;=============================================================================
; Misc-functions:
;
; The fixed point combinator:
;
(define (fix f) (lambda (x) ((f (fix f)) x)))
;
(define (fcl a sc)
  (if (zero? a)
      (sc '())
      (lambda (v)
        (fcl (sub1 a) (lambda (l) (sc (cons v l)))))))
;
; Making environments static:
;
(define (make-env-static namelist env envi)
  (lambda (name)
    (dynamic-lookup name namelist env envi)))
;
(define (dynamic-lookup name namelist env envi)
  (if (null? namelist)
      (envi name)
      (let ([s-name (car namelist)])
        (if (equal? name s-name)
            (env s-name)
            (dynamic-lookup name (cdr namelist) env envi)))))
;
; Environments:
;
(define (upd i v r)
  (lambda (w)
    (if (equal? w i) (my-force v) (r w))))
;
(define (init-venv)
  (lambda (w) (error '*** "Unknown variable: ~s" w)))
;
(define (init-fenv)
  (lambda (w) (library-call w)))
;
(define (init-cenv)
  (lambda (cn)
    (if (equal? cn '!cons)
        (lambda (x) (lambda (y) (inV: x y)))
        (error '*** "Unknown constructor: ~s" cn))))
```

Appendix D

The Final Interpreter

```
; -*- Scheme -*- mode
;-------------------------------------------------------------------------
; File : bawl.sim. Final version with on-line force optimization.
;
; An interpreter for the language Bawl intended to be specialized by Similix.
;
; July 1991.  - Jesper J@orgensen -
;-------------------------------------------------------------------------
; Variable names:
;
;   Environments:
;
;     venv :: Variable-environment == Variable->Delayed-value
;     oenv :: Occurrence-environment == Occurrence->Delayed-value
;    voenv :: Variable-2-occurrence-environment == Variable->Occurrence
;     fenv :: Function-environment
;     cenv :: constructor-environment
;
;   Syntactic variables:
;
;      scr :: Script == [Definition]
;      def :: Definition
;       eq :: Equations
;     expr :: Expression
;      pat :: Pattern
;      alt :: Alternative
;    guard :: Expression
;      elc :: ListComprehension
;     qual :: Qualifier
;
;   Values:
;
;        d :: Delayed-value
;        v :: Value
;
;   Continuations:
;
;       sc :: Success-continuation
;       fc :: Failure-continuation
;        c :: Continuation
;
;   Tables:
;
;      cfs :: [User-defined-conformals]
;     ctab :: [Constructor-families]
;
;   Miscellaneous:
;
;      occ :: Occurrence
;       uo :: Used-occurrences == [Occurrence]
```

```
;       ds :: Description
;       fv :: Freevariables == [variable]
;       dv :: DefinedVariables == [variable]
;       fo :: ForceVariable == [variable]
;      a,n :: Integer
;
;
; Variable names may have prefixes or postfixes attached and the list above
; is just intended to be a guideline for understanding the meaning of
; variable names in the program. A postfix * means "list of".
;
; The names of functions in the interpreter corresponds to the names in the
; semantics.
;
;------------------------------------------------------------------------
(load "bawl-work.syn")
(loadt "bawl-work.adt")
(loadt (string-append **Similix-library** "scheme.adt"))
;========================================================================
; Main function:
;
;   scr   : script
;   fenvi : initial function environment
;   cenvi : initial constructor environment
;
(define (S scr)
  (let* ([scr  (elab-scr scr)]
         [funs (collect-functions scr)]
         [ctab (make-ctab scr)]
         [cenv (CE scr)]
         [fenv (FE scr (compile-time-init-venv) cenv (init-fenv) ctab)])
    (vector
     (make-env-static (collect-constructors scr) cenv (init-cenv))
     (make-env-static  funs fenv (init-fenv))
     funs)))
;
;========================================================================
; Constructor environment:
;
(define (CE def*)
  (if (null? def*)
      (init-cenv)
      (let ([def1 (def*->def1 def*)])
        (if (TypeDef? def1)
            (CD (def->cdef* def1) (CE (def*->def* def*)))
            (CE (def*->def* def*))))))
;
(define (CD cdef* cenv)
  (if (null? cdef*)
      cenv
      (let* ([cdef (cdef*->cdef1 cdef*)]
             [con  (cdef->con cdef)]
             [a    (length (cdef->atexpr* cdef))])
        (upd con (fcl-c a (lambda (vs)
                            (inVCon con (listToTuple vs))))
             (CD (cdef*->cdef* cdef*) cenv)))))
;
;========================================================================
; Function environment:
;
(define (FE scr venv cenv fenvi ctab)
  (let ([ncfs (collect-conformals scr)]) ; New conformals at this level.
    (if (null? ncfs)
        (fix
         (lambda (fenv)
           (lambda (name)
             (FD scr name venv cenv fenv fenvi (vector) ncfs ctab))))
```

```
        (let ([cfst (make-cfst (length ncfs))])
          (fix
           (lambda (fenv)
             (lambda (name)
               (FD scr name venv cenv fenv fenvi cfst ncfs ctab))))))))

(define (FD def* name venv cenv fenv fenvi cfst ncfs ctab)
  (if (null? def*)
      (fenvi name)
      (let* ([def1    (def*->def1 def*)]
             [newdef* (def*->def* def*)])
        (if (and (FunDef? def1) (equal? name (def->fun def1)))
            (restrict-fenv
             (freefunsdef def1)
             fenv
             (lambda (fenv1)
               (let* ([eq* (def->eq* def1)]
                      [a   (Funarity eq*)])
                 (if (zero? a)
                     (save-at
                      (conf-store-ref cfst (location name ncfs))
                      (lambda ()
                        (ME a eq* (init-oenv) (makeDbottomlist a) '() '()
                            venv cenv fenv1 name ctab)))
                     (fcl-f 0 a
                            (lambda (oenv)
                              (ME a eq* oenv (makeDbottomlist a) '() '()
                                  venv cenv fenv1 name ctab)))))))
            (FD newdef* name venv cenv fenv fenvi cfst ncfs ctab)))))
;
;---------------------------------------------------------------------------
; Match equations:
;
(define (ME a eq* oenv ds* uo fo venv cenv fenv name ctab)
  (if (null? eq*)
      (error '*** "No matching equations for function: ~s" name)
      (let* ([eq (eq*->eq1 eq*)]
             [pat* (eq->pat* eq)]
             [alt* (eq->alt* eq)]
             [def* (eq->def* eq)])
        (MP 0 a pat* oenv (inOTop) ds* uo fo ctab (init-voenv)
            (lambda (voenv1 ds1* oenv1 uo1 fo1)
              (let* ([fvalt* (freevaralt* alt*)]
                     [fvdef* (freevardef* def*)]
                     [fv     (union fvdef* fvalt*)]
                     [venv1  (make-new-venv fv voenv1 oenv1 uo1 fo1 venv)]
                     [fo2    (find-new-fov fv voenv1 uo1 fo1)])
                (MA alt* (restrict-venv fvalt* venv1) cenv
                    (if (null? def*)
                        fenv
                        (FE def* (restrict-venv-and-suspend fvdef* venv1 fo2)
                            cenv fenv ctab))
                    name ctab fo2)))
            (lambda (ds1* oenv1 uo1 fo1)
              (ME a (eq*->eq* eq*) oenv1 ds1* uo1 fo1 venv cenv fenv name ctab))))))
;
;---------------------------------------------------------------------------
; Match alternatives:
;
(define (MA alt* venv cenv fenv name ctab fo)
  (if (null? alt*)
      (error '*** "No matching alternatives for function: ~s" name)
      (let* ([alt   (alt*->alt alt*)]
             [expr  (alt->expr alt)]
             [guard (alt->guard alt)])
        (checkguard expr guard (alt*->alt* alt*) venv cenv fenv name ctab fo))))
;
```

```
;=============================================================================
; Evaluation of expressions:
;
(define (E expr venv cenv fenv ctab fo c)
  (cond
   ((ELit?   expr) (c (L (ELit->lit expr)) fo venv))
   ((EVar?   expr) (let ([var (EVar->var expr)])
                     (if (member var fo)
                         (c (venv var) fo venv)
                         (let ([v (my-force (venv var))])
                           (c v (cons var fo) (upd var v venv))))))
   ((EFun?  expr)  (c (fenv (EFun->fun expr)) fo venv))
   ((EOp?   expr)  (c (Op0 expr) fo venv))
   ((EBool?  expr) (c (Bool expr) fo venv))
   ((ENil?   expr) (c (inVNil) fo venv))
   ((E:?     expr) (c (inV: (EL (E:->Expr1 expr) venv cenv fenv ctab fo)
                            (EL (E:->Expr2 expr) venv cenv fenv ctab fo))
                      fo venv))
   ((ECon?  expr)  (c (cenv (ECon->con expr)) fo venv))
   ((ETuple? expr) (c (Tuple (ETuple->expr* expr) venv cenv fenv ctab fo) fo venv))
   ((EApply? expr)
    (cond
     ((EOp1?  expr) (Op1 expr venv cenv fenv ctab fo c))
     ((EOp2?  expr) (Op2 expr venv cenv fenv ctab fo c))
     (else           (E (EApply->Expr1 expr) venv cenv fenv ctab fo
                        (lambda (f fo1 venv1)
                          (c (f (EL (EApply->Expr2 expr) venv1 cenv fenv ctab fo1))
                             fo1 venv1))))))
   ((EAS?    expr) (AS expr venv cenv fenv ctab fo c))
   ((ELC?    expr) (LC expr cenv fenv venv (suspend (inVNil)) ctab fo))
   (else
    (compile-time-error '*** "Syntax error in expreesion: ~s" expr))))

(define (EL expr venv cenv fenv ctab fo)
  (cond
   ((EVar? expr)
    (let ([var (EVar->var expr)])
      (if (member var fo)
          (suspend (venv (EVar->var expr)))
          (venv (EVar->var expr)))))
   ((or (EApply? expr) (EAS? expr) (ELC? expr) (E:? expr) (ETuple? expr))
    (my-delay (E expr venv cenv fenv ctab fo idc)))
   (else
    (suspend (E expr venv cenv fenv ctab fo idc)))))
;
;=============================================================================
;  Evaluation of litarals:
;
(define (L lit)
  (cond
   ((LNum? lit)  (Lit->num lit))
   ((LChar? lit) (Lit->char lit))
   (else ; lit a string
    (lit->String lit))))
;
;=============================================================================
; Evaluation of tuples:
;
(define (Tuple expr* venv cenv fenv ctab fo)
  (let ([a (length expr*)])
    (cond
     ((= a 0) (inVTuple0 0))
     ;(= a 1) 1-Tuples does not exist!
     ((= a 2) (inVTuple2 2
                (EL (car expr*) venv cenv fenv ctab fo)
                (EL (cadr expr*) venv cenv fenv ctab fo)))
     ((= a 3) (inVTuple3 3
```

```
               (EL (car expr*) venv cenv fenv ctab fo)
               (EL (cadr expr*) venv cenv fenv ctab fo)
               (EL (caddr expr*) venv cenv fenv ctab fo)))
     (else    (inVTuple* a
               (MEL expr* venv cenv fenv ctab fo))))))

(define (MEL expr* venv cenv fenv ctab fo)
  (if (null? expr*)
      '()
      (cons (EL (car expr*) venv cenv fenv ctab fo)
            (MEL (cdr expr*) venv cenv fenv ctab fo))))
;
;============================================================================
; Evaluation of basic operators:
;
; Basic operators not applied to arguments:
;
(define (Op0 expr)
  (let* ([op (EOp->Op expr)])
    (lambda (d1)
      (cond
       ((Binop? op)
        (lambda (d2)
          (cond
           ((Compose? op)
            (lambda (d3) ((my-force d1) (my-delay ((my-force d2) d3)))))
           ((Lazybinop? op)
            (cond
             ((equal? op '!append) (lazy-append d1 d2))
             (else ; op '!listdiff
              (lazy-list-subtract d1 d2))))
           ((StrictLazybinop? op)
            (let ([v1 (my-force d1)])
              (cond
               ((equal? op '!and)
                (if v1 (my-force d2) #f))
               (else ;op = !or
                (if v1 #t (my-force d2))))))
           (else ; (Strictbinop? op)
            (StrictBinop op (my-force d1) (my-force d2))))))
       ((Uop? op)
        (if (lazyuop? op)            ; op = !length
            (lazy-length d1)
            (let ([v1 (my-force d1)]) ; (Strictuop? op)
              (cond
               ((equal? op '!not) (not v1))
               (else ; op = !uminus
                (- 0 v1))))))
       (else (compile-time-error '*** "Unknown basis-operator: ~s" op))))))
;
;----------------------------------------------------------------------------
; Basic operators applied to 1 argument:
;
(define (Op1 expr venv cenv fenv ctab fo c)
  (let* ([op (EOp->Op (Eapply->expr1 expr))]
         [expr1 (Eapply->expr2 expr)])
    (cond
     ((Binop? op)
      (c (lambda (d2)
           (cond
            ((Compose? op)
             (lambda (d3)
               ((E expr1 venv cenv fenv ctab fo idc)
                (my-delay ((my-force d2) d3)))))
            ((Lazybinop? op)
             (let ([d1 (EL expr1 venv cenv fenv ctab fo)])
               (cond
```

```
               ((equal? op '!append) (lazy-append d1 d2))
               (else ; op '!listdiff
                (lazy-list-subtract d1 d2)))))
           ((StrictLazybinop? op)
            (let ([v1 (E expr1 venv cenv fenv ctab fo idc)])
              (cond
               ((equal? op '!and)
                (and v1 (my-force d2)))
               (else ; op = !or
                (or v1 (my-force d2))))))
           (else ; (Strictbinop? op)
            (StrictBinop op (E expr1 venv cenv fenv ctab fo idc) (my-force d2)))))
        fo venv))
     ((Uop? op)
      (if (lazyuop? op) ; op = !length
          (c (lazy-length (EL expr1 venv cenv fenv ctab fo)) fo venv)
          (E expr1 venv cenv fenv ctab fo ; (Strictuop? op)
             (lambda (v1 fo1 venv1)
               (c (cond
                   ((equal? op '!not) (not v1))
                   (else ; op = !uminus
                    (- 0 v1)))
                  fo1 venv1)))))
     (else (compile-time-error '*** "Unknown basis-operator: ~s" op)))))
;
;-----------------------------------------------------------------------------
; Basic operators applied to 2 arguments:
;
(define (Op2 expr venv cenv fenv ctab fo c)
  (let* ([expr3 (Eapply->expr1 expr)]
         [expr2 (Eapply->expr2 expr)]
         [op (EOp->Op (Eapply->expr1 expr3))]
         [expr1 (Eapply->expr2 expr3)])
    (cond
     ((Compose? op)
      (c (lambda (d3)
           ((E expr1 venv cenv fenv ctab fo idc)
            (my-delay ((E expr2 venv cenv fenv ctab fo idc) d3))))
         fo venv))
     ((Lazybinop? op)
      (let* ([d1 (EL expr1 venv cenv fenv ctab fo)]
             [d2 (EL expr2 venv cenv fenv ctab fo)])
        (c (cond
            ; List operations:
            ((equal? op '!append) (lazy-append d1 d2))
            (else ; op = !listdiff
             (lazy-list-subtract d1 d2)))
           fo venv)))
     ((StrictLazybinop? op)
      (E expr1 venv cenv fenv ctab fo
         (lambda (v1 fo1 venv1)
           (c (cond
               ; Logical operations:
               ((equal? op '!and)
                (and v1 (E expr2 venv1 cenv fenv ctab fo1 idc)))
               (else ; op = '!or
                (or v1 (E expr2 venv1 cenv fenv ctab fo1 idc))))
              fo1 venv1))))
     ((Strictbinop? op)
      (E expr1 venv cenv fenv ctab fo
         (lambda (v1 fo1 venv1)
           (E expr2 venv1 cenv fenv ctab fo1
            (lambda (v2 fo2 venv2)
              (c (StrictBinop op v1 v2) fo2 venv2))))))
     (else (compile-time-error '*** "Unknown basis-operator: ~s" op)))))

(define (StrictBinop op v1 v2)
```

```
  (cond
   ((equal? op '!index) (lazy-index v1 v2))
   ((equal? op '!greater) (> v1 v2))
   ((equal? op '!notless) (>= v1 v2))
   ((equal? op '!equal) (struct-equal? v1 v2))
   ((equal? op '!notequal) (not-equal? v1 v2))
   ((equal? op '!notgreater) (<= v1 v2))
   ((equal? op '!less) (< v1 v2))
   ((equal? op '!plus) (+ v1 v2))
   ((equal? op '!minus) (- v1 v2))
   ((equal? op '!times) (* v1 v2))
   ((equal? op '!divide) (/ v1 v2))
   ((equal? op '!div) (round (/ v1 v2)))
   ((equal? op '!mod) (modulo v1 v2))
   ((equal? op 'strict) (strict v1 v2))
   (else ; op = !power
    (power v1 v2))))
;
;============================================================================
; Evaluate arithmetic sequences:
;
(define (AS expr venv cenv fenv ctab fo c)
  (let* ([expr1 (Eas->Expr1 expr)]
         [expr2 (Eas->Expr2 expr)]
         [expr3 (Eas->Expr3 expr)])
    (if (ENone? expr3)
        (E expr1 venv cenv fenv ctab fo
           (lambda (v1 fo1 venv1)
             (if (ENone? expr2)
                 (c (from v1) fo1 venv1)
                 (E expr2 venv1 cenv fenv ctab fo1
                    (lambda (v2 fo2 venv2)
                      (c (fromThen v1 (- v2 v1)) fo2 venv2))))))
        (E expr1 venv cenv fenv ctab fo
           (lambda (v1 fo1 venv1)
             (E expr3 venv1 cenv fenv ctab fo1
                (lambda (v3 fo3 venv3)
                  (if (ENone? expr2)
                      (c (fromTo v1 v3) fo3 venv3)
                      (E expr2 venv3 cenv fenv ctab fo3
                         (lambda (v2 fo2 venv2)
                           (let ([s (- v2 v1)])
                             (if (>= s 0)
                                 (c (fromUpTo v1 s v3) fo2 venv2)
                                 (c (fromDownTo v1 s v3) fo2 venv2))))))))))))
;
;============================================================================
; Evaluate list comprehension:
;
(define (LC elc cenv fenv venv vs ctab fo)
  (let* ([expr0 (ELC->expr elc)]
         [qual* (ELC->qual* elc)])
    (if (null? qual*)
        (inV: (EL expr0 venv cenv fenv ctab fo) vs)
        (let* ([qual (car qual*)]
               [qual1* (cdr qual*)])
          (if (LCgen? qual)
              (E (LCgen->expr qual) venv cenv fenv ctab fo
                 (lambda (v1 fo1 venv1)
                   (let* ([elc1  (inLC expr0 qual1*)]
                          [fv    (freevarexp elc1 '())]
                          [venv2 (restrict-venv fv venv1)])
                     (G (LCgen->pat qual)
                        (lambda (voenv oenv uo fo3 v3)
                          (let* ([fo4   (union fo3 (intersection fo1 fv))]
                                 [venv4 (make-new-venv fv voenv oenv uo fo4 venv2)])
                            (LC elc1 cenv fenv venv4 v3 ctab fo4)))
```

```
                    vs v1 ctab))))
            (E (LCfilter->expr qual) venv cenv fenv ctab fo
               (lambda (t fo1 venv1)
                 (let* ([elc1  (inLC expr0 qual1*)]
                        [fv    (freevarexp elc1 '())]
                        [venv2 (restrict-venv fv venv1)])
                   (if t
                       (LC (inLC expr0 qual1*) cenv fenv venv2 vs ctab fo1)
                       (my-force vs)))))))))

(define (G pat c vs1 vs ctab)
  (if (VNil? vs)
      (my-force vs1)
      (let* ([v1 (car vs)]
             [occ (inOarg 1 (inOTop))])
        (P pat (upd-o occ v1 (init-oenv)) occ
           (newDBottom) '() '() ctab (init-voenv)
           (lambda (voenv ds oenv uo fo1)
             (c voenv oenv uo fo1
                (suspend
                 (G pat c vs1 (my-force (cdr vs)) ctab))))
           (lambda (ds oenv uo fo)
             (G pat c vs1 (my-force (cdr vs)) ctab))))))
;
;=============================================================================
; Pattern matching :
;
(define (MP n a pat* oenv occ ds* uo fo ctab voenv sc fc)
  (if (null? pat*)
      (apply-fun n a (sc voenv ds* oenv uo fo) oenv)
      (let ([p1 (pat*->p1 pat*)])
        (P p1 oenv (inOArg n occ) (car ds*) uo fo ctab voenv
           (lambda (voenv1 ds1 oenv1 uo1 fo1)
             (MP (add1 n) a (pat*->pat* pat*) oenv1 occ
                 (cdr ds*) uo1 fo1 ctab voenv1
                 (lambda (voenv2 ds2* oenv2 uo2 fo2)
                   (sc voenv2 (cons ds1 ds2*) oenv2 uo2 fo2))
                 (lambda (ds2* oenv2 uo2 fo2)
                   (fc (cons ds1 ds2*) oenv2 uo2 fo2))))
           (lambda (ds1 oenv1 uo1 fo1)
             (fc (cons ds1 (cdr ds*)) oenv1 uo1 fo1))))))

(define (P pat oenv occ ds uo fo ctab voenv sc fc)
  (cond
   ((PVar? pat) (let* ([var  (Pvar->var pat)]       ; Check if variable is all
                       [occ1 (lookup-s var voenv)]) ; ready bound.
                  (if (Free? occ1)                  ; => Variable not bound
                      (sc (upd-s var occ voenv) ds oenv uo fo)
                      (occ-lookup-s
                       oenv occ uo fo
                       (lambda (v1 uo1 fo1 oenv1)
                         (occ-lookup-s
                          oenv1 occ1 uo1 fo1
                          (lambda (v2 uo2 fo2 oenv2)
                            (if (struct-equal? v1 v2)
                                (sc voenv ds oenv2 uo2 fo2)
                                (fc ds oenv2 uo2 fo2)))))))))
   ((Dbot? ds)  (Pbot pat oenv occ uo fo ctab voenv sc fc))
   ((Dlit? ds)  (Plit pat oenv ds uo fo voenv sc fc))
   ((Dbool? ds) (Pbool pat oenv ds uo fo voenv sc fc))
   ((Dcon? ds)  (Pcon pat oenv occ ds uo fo ctab voenv sc fc))
   ((Dpair? ds) (Ppair pat oenv occ ds uo fo ctab voenv sc fc))
   ((DNil? ds)  (Pnil pat oenv ds uo fo voenv sc fc))
   ((DTuple? ds)(PTuple pat oenv occ ds uo fo ctab voenv sc fc))
   ((DNlit? ds) (PNlit pat oenv occ ds uo fo voenv sc fc))
   (else ; (DNcon? ds)
    (PNcon pat oenv occ ds uo fo ctab voenv sc fc))))
```

```
(define (Pbot pat oenv occ uo fo ctab voenv sc fc)
  (if (or (PTuple? pat)
          (and (Pcon? pat)
               (unaryfamily (Pcon->con pat) ctab)))
      (Pbot-lazy pat oenv occ uo fo ctab voenv sc fc)
      (Pbot-strict pat oenv occ uo fo ctab voenv sc fc)))

(define (Pbot-strict pat oenv occ uo fo ctab voenv sc fc)
  (occ-lookup-s
   oenv occ uo fo
   (lambda (fd uo1 fo1 oenv1)
     (cond
      ((Plit? pat)
       (let ([lit (Plit->lit pat)])
         (if (equal? (L lit) fd)
             (sc voenv (inDlit lit) oenv1 uo1 fo1)
             (fc (newDNlit lit) oenv1 uo1 fo1))))
      ((Pbool? pat)
       (if (Pbool->bool pat)
           (if fd
               (sc voenv (inDbool #t) oenv1 uo1 fo1)
               (fc (inDBool #f) oenv1 uo1 fo1))
           (if fd
               (fc (inDBool #t) oenv1 uo1 fo1)
               (sc voenv (inDbool #f) oenv1 uo1 fo1))))
      ((Pcon? pat)
       (let ([con (Pcon->con pat)]
             [dcon (Vcon->con fd)])
         (if (equal? con dcon)
             (Pcon pat (upd-o (inOConName occ) dcon oenv1) occ
                   (newDcon con (arity-con con ctab))
                   uo1 fo1 ctab voenv sc fc)
             (fc (newDNcon con ctab)
                 (upd-o (inOConName occ) dcon oenv1) uo1 fo1))))
      ((Ppair? pat)
       (if (V:? fd)
           (Ppair pat oenv1 occ (newDpair) uo1 fo1 ctab voenv sc fc)
           (fc (newDnil) oenv1 uo1 fo1)))
      ((Pnil? pat)
       (if (VNil? fd)
           (sc voenv (newDnil) oenv1 uo1 fo1)
           (fc (newDpair) oenv1 uo1 fo1)))
      (else
       (compile-time-error '*** "Syntax error in pattern: ~s" pat))))))

(define (Pbot-lazy pat oenv occ uo fo ctab voenv sc fc)
  (cond
   ((PTuple? pat)
    (PTuple pat oenv occ (NewDTuple (PTuple->size pat)) uo fo ctab voenv sc fc))
   (else
    (let ([con (Pcon->con pat)])
      (Pcon pat oenv occ (newDcon con (arity-con con ctab))
            uo fo ctab voenv sc fc)))))

(define (Plit pat oenv ds uo fo voenv sc fc)
  (if (equal? (Plit->lit pat) (Dlit->lit ds))
      (sc voenv ds oenv uo fo)
      (fc ds oenv uo fo)))

(define (Pbool pat oenv ds uo fo voenv sc fc)
  (if (equal? (Dbool->bool ds) (Pbool->bool pat))
      (sc voenv ds oenv uo fo)
      (fc ds oenv uo fo)))

(define (Pcon pat oenv occ ds uo fo ctab voenv sc fc)
  (let ([con (Pcon->con pat)])
```

```
    (if (equal? con (Dcon->con ds))
        (let* ([pat* (Pcon->pat* pat)]
               [a (length pat*)]
               [ds* (Dcon->ds* ds)])
          (MP 0 a pat* oenv (inOCon occ) ds* uo fo ctab voenv
              (lambda (voenv1 ds1* oenv2 uo2 fo2)
                (sc voenv1 (inDcon con ds1*) oenv2 uo2 fo2))
              (lambda (ds1* oenv2 uo2 fo2)
                (fc (inDcon con ds1*) oenv2 uo2 fo2))))
        (fc ds oenv uo fo))))

(define (Ppair pat oenv occ ds uo fo ctab voenv sc fc)
  (if (Ppair? pat)
      (P (Ppair->p1 pat) oenv (inOHead occ) (Dpair->D1 ds) uo fo ctab voenv
         (lambda (voenv1 ds1 oenv1 uo1 fo1)
           (P (Ppair->p2 pat) oenv1 (inOTail occ) (Dpair->D2 ds)
              uo1 fo1 ctab voenv1
              (lambda (voenv2 ds2 oenv2 uo2 fo2)
                (sc voenv2 (inDpair ds1 ds2) oenv2 uo2 fo2))
              (lambda (ds2 oenv2 uo2 fo2)
                (fc (inDpair ds1 ds2) oenv2 uo2 fo2))))
         (lambda (ds1 oenv1 uo1 fo1)
           (fc (inDpair ds1 (Dpair->D2 ds)) oenv1 uo1 fo1)))
      (fc ds oenv uo fo)))

(define (Pnil pat oenv ds uo fo voenv sc fc)
  (if (Pnil? pat)
      (sc voenv ds oenv uo fo)
      (fc ds oenv uo fo)))

(define (PTuple pat oenv occ ds uo fo ctab voenv sc fc)
  (MP 0 (Ptuple->size pat) (PTuple->pat* pat) oenv (inOTuple occ)
      (Dtuple->ds* ds) uo fo ctab voenv
      (lambda (voenv1 ds1* oenv1 uo1 fo1)
        (sc voenv1 (inDTuple ds1*) oenv1 uo1 fo1))
      (lambda (ds1* oenv1 uo1 fo1)
        (fc (inDTuple ds1*) oenv1 uo1 fo1))))

(define (PNlit pat oenv occ ds uo fo voenv sc fc)
  (let* ([lit (Plit->lit pat)])
    (if (member lit (DNlit->set ds))
        (fc ds oenv uo fo)
        (occ-lookup-s
         oenv occ uo fo
         (lambda (v uo1 fo1 oenv1)
           (if (equal? (L lit) v)
               (sc voenv (inDlit lit) oenv1 uo1 fo1)
               (fc (addDNlit lit ds) oenv1 uo1 fo1)))))))

(define (PNcon pat oenv occ ds uo fo ctab voenv sc fc)
  (let* ([con  (Pcon->con pat)]
         [cset (DNcon->set ds)])
    (if (member con cset)
        (if (equal? (length cset) 1)
            (Pcon pat oenv occ (newDcon con (arity-con con ctab))
                  uo fo ctab voenv sc fc)
            (if (equal? con (oenv (inOConName occ)))
                (Pcon pat oenv occ (newDcon con (arity-con con ctab))
                      uo fo ctab voenv sc fc)
                (fc (delDNCon con ds) oenv uo fo)))
        (fc ds oenv uo fo))))
;
;-----------------------------------------------------------------------
; Evaluate guard after successfull matching of alternative:
;
(define (checkguard expr guard alt* venv cenv fenv name ctab fo)
  (if (TrueGuard? guard)
```

```
      (E expr venv cenv fenv ctab fo idc)
      (E guard venv cenv fenv ctab fo
         (lambda (t fo1 venv1)
           (if t
               (E expr venv1 cenv fenv ctab fo1 idc)
               (MA alt* venv1 cenv fenv name ctab fo1))))))
;
;============================================================================
; Miscellaneous-functions:
;
; The fixpoint operator:
(define (fix f)
  (lambda (x) ((f (fix f)) x)))
;
;----------------------------------------------------------------------------
; Make closures:
;
(define (fcl-c a sc)
  (if (zero? a)
      (sc '())
      (lambda (v)
        (fcl-c (sub1 a) (lambda (l) (sc (cons v l)))))))

(define (fcl-f n a sc)
  (if (= n a)
      (sc (init-oenv))
      (lambda (v)
        (fcl-f (add1 n) a
               (lambda (r) (sc (upd-o (inOarg n (inOTop)) v r)))))))
;
;----------------------------------------------------------------------------
; Apply expression to rest of arguments after pattern matching:
;
(define (apply-fun n a f er)
  (if (= n a)
      f
      (apply-fun (add1 n) a (f (er n)) er)))
;
;----------------------------------------------------------------------------
; Lookup the value of a dynamic name known to be one of the elements in
; the static list namelist:
;
(define (make-env-static namelist env envi)
  (lambda (name) (dynamic-lookup name namelist env envi)))
;
(define (dynamic-lookup name namelist env envi)
  (if (null? namelist)
      (envi name)
      (let ([s-name (car namelist)])
        (if (equal? name s-name)
            (env s-name)
            (dynamic-lookup name (cdr namelist) env envi)))))
;
;----------------------------------------------------------------------------
; Find the value pointed to by occ. Update oenv with value of occ and uo
; with the information that occ has been used. Finaly, the continuation c
; is applied to the value, oenv and uo. If version = s then the resulting
; value will be evaluated completely and the result is dynamic. If version
; = l then no more evaluation take place and the result is closure.
;
; version = s (strict) - returns a dynamic result!
(define (occ-lookup-s oenv occ uo fo c)
  (let* ((pooc (cdr occ)))
    (if (or (member occ uo) (OTop? pooc))
        (let ([uo1 (if (OTop? pooc) (cons occ uo) uo)])
          (if (not (member occ fo))
              (let ([v (my-force (oenv occ))])
```

```
                (c v uo1 (cons occ fo) (upd-o occ v oenv)))
              (c (oenv occ) uo1 fo oenv)))
        (occ-lookup-s
         oenv pooc uo fo
         (lambda (v/d1 uo1 fo1 oenv1)
           (let* ([v/d (cond
                        ((OConArg? occ)
                         (VCon-ref v/d1 (add1 (OConArg->arg occ))))
                        ((OTupleArg? occ)
                         (VTuple-ref v/d1 (add1 (OTupleArg->arg occ))))
                        ((O:head? occ)
                         (car v/d1))
                        (else
                         (cdr v/d1)))]
                  [uo2 (cons occ uo1)]
                  [fo2 (cons occ fo1)])
             (c v/d uo2 fo2 (upd-o occ v/d oenv1)))))))
;
; version = l (lazy) returns a closure result!
(define (occ-lookup-l oenv occ uo fo c)
  (let* ((pooc (cdr occ)))
    (if (or (member occ uo) (OTop? pooc))
        (let ([uo1 (if (OTop? pooc) (cons occ uo) uo)])
          (c (oenv occ) uo1 fo oenv))
        (occ-lookup-l
         oenv pooc uo fo
         (lambda (v/d1 uo1 fo1 oenv1)
           (let* ([d (if (member pooc fo)
                         (cond
                          ((OConArg? occ)
                           (VCon-ref v/d1 (add1 (OConArg->arg occ))))
                          ((OTupleArg? occ)
                           (VTuple-ref v/d1 (add1 (OTupleArg->arg occ))))
                          ((O:head? occ)
                           (car v/d1))
                          (else
                           (cdr v/d1)))
                         (suspend
                          (my-force
                           (let ([v1 (my-force v/d1)])
                             (cond
                              ((OConArg? occ)
                               (VCon-ref v1 (add1 (OConArg->arg occ))))
                              ((OTupleArg? occ)
                               (VTuple-ref v1 (add1 (OTupleArg->arg occ))))
                              ((O:head? occ)
                               (car v1))
                              (else
                               (cdr v1))))))))]
                  [uo2 (cons occ uo1)])
             (c d uo2 fo1 (upd-o occ d oenv1)))))))
;
;---------------------------------------------------------------------------
; Make a new variable-environment from
;
;   an occurrence-environment   : oenv
;   an variable-occurrences-env.: voenv
;   used occurrences            : uo
;   free variable in expression : fv
;
; only binding the free variables:
;
(define (make-new-venv fv voenv oenv uo fo venv)
   (make-new-venv-c fv voenv oenv uo fo venv (compile-time-init-venv)))

(define (make-new-venv-c fv voenv oenv uo fo venv new-venv)
  (if (null? fv)
```

```
      new-venv
      (let* ((v1    (car fv))
             (occ   (lookup-s v1 voenv)))
        (if (Free? occ) ; Variable not bound by pattern matching!
            (make-new-venv-c (cdr fv) voenv oenv uo fo venv
                             (upd v1 (venv v1) new-venv))
            (occ-lookup-l oenv occ uo fo
                          (lambda (d uo1 fo1 oenv1)
                            (make-new-venv-c (cdr fv) voenv oenv1 uo1 fo venv
                                             (upd v1 d new-venv))))))))
;
;---------------------------------------------------------------------------
; Restrict the size of environments:
;
(define (restrict-venv fv venv)
  (if (null? fv)
      (compile-time-init-venv)
      (let ([n (car fv)])
        (upd n (venv n) (restrict-venv (cdr fv) venv)))))

(define (restrict-venv-and-suspend fv venv fov)
  (if (null? fv)
      (compile-time-init-venv)
      (let ([n (car fv)])
        (upd n (if (member n fov) (suspend (venv n)) (venv n))
             (restrict-venv-and-suspend (cdr fv) venv fov)))))

(define (restrict-fenv funs fenv c)
  (if (null? funs)
      (c (init-fenv))
      (let ([f (car funs)])
        (restrict-fenv (cdr funs) fenv
                       (lambda (r)
                         (c (upd f (fenv f) r)))))))
;
;---------------------------------------------------------------------------
; Initial environments:
;
(define (compile-time-init-venv)
  (lambda (w)
    (compile-time-error '*** "Unknown variable: ~s" w)))
;
(define (init-oenv)
  (lambda (w) '()))
;
(define (init-fenv)
  (lambda (w) (library-call w)))
;
(define (init-cenv)
  (lambda (w) (standard-constructor w)))
;
;---------------------------------------------------------------------------
; Update environments:
;
(define (upd-o u v r)
  (lambda (w)
    (if (equal? w u)
        v
        (r w))))
;
;---------------------------------------------------------------------------
; Identity continuation:
;
(define (idc v fo venv) v)
```

Appendix E

The Adt-files for the Final Interpreter

```
; -*- Scheme -*- mode
;
; Bawl-final.adt - New version April. - 1991. - Jesper J@orgensen.
;
;------------------------------------------------------------------------------
; Expression sort
;
;   Expr -> (lit Lit) | (var Var) | (fun Fun) | (con Con) | (op Op) |
;           (nil) | (pair Expr Expr) | (tuple Expr*) | (apply Expr Expr) |
;           (from Expr Expr Expr) | (lcomp Expr Qual*)
;
; Predicates:
;
(defprim (Elit? exp)
  (and (pair? exp) (equal? (car exp) 'lit)))
(defprim (Evar? exp)
  (and (pair? exp)
       (equal? (car exp) 'var)))
(defprim (Econ? exp)
  (and (pair? exp)
       (equal? (car exp) 'cons)
       (not (member (Econ->con exp) '(|True| |False|)))))
(defprim (Efun? exp)
  (and (pair? exp)
       (equal? (car exp) 'fun)))
(defprim (EOp? exp)
  (and (pair? exp)
       (equal? (car exp) 'op)))
(defprim (Enil? exp) (equal? exp '(nil)))
(defprim
   (E:? exp)
   (and (pair? exp) (equal? (car exp) 'pair)))
(defprim (Etuple? exp)
  (and (pair? exp) (equal? (car exp) 'tuple)))
(defprim (Eapply? exp)
  (and (pair? exp) (equal? (car exp) 'apply)))
(defprim (EAS? exp)
  (and (pair? exp) (equal? (car exp) 'from)))
(defprim (Elc? exp)
  (and (pair? exp) (equal? (car exp) 'lcomp)))
;
; Selectors:
;
(defprim 1 Elit->lit cadr)
(defprim 1 Evar->var cadr)
(defprim 1 Econ->con cadr)
(defprim 1 Efun->fun cadr)
```

```
(defprim 1 EOp->op cadr)
(defprim 1 E:->expr1 cadr)
(defprim 1 E:->expr2 caddr)
(defprim 1 Eapply->expr1 cadr)
(defprim 1 Eapply->expr2 caddr)
(defprim 1 Etuple->expr* cadr)
(defprim 1 Eas->expr1 cadr)
(defprim 1 Eas->expr2 caddr)
(defprim 1 Eas->expr3 cadddr)
(defprim 1 Elc->expr cadr)
(defprim 1 Elc->qual* caddr)
(defprim 1 Exp*->expr1 car)
(defprim 1 Exp*->tl cdr)
;
; Injections:
;
(defprim (inELit lit)  (list 'lit lit))
(defprim (inEvar name) (list 'var name))
(defprim (inEcon name) (list 'cons name))
(defprim (inEfun name) (list 'fun name))
(defprim (inEOp name)  (list 'op name))
(defprim (inE: e1 e2) (list 'pair e1 e2))
(defprim (inEApply e1 e2) (list 'apply e1 e2))
(defprim (inETuple e*) (list 'tuple e*))
(defprim (inEAS e1 e2 e3) (list 'from e1 e2 e3))
(defprim (inLC e q) (list 'lcomp e q))
;
; Special cases:
;
(defprim (Ebool? exp)
  (or (equal? exp '(cons |True|))
      (equal? exp '(cons |False|))))
(defprim (EOp1? exp)
  (and (Eapply? exp)
       (EOp? (Eapply->expr1 exp))))
(defprim (EOp2? exp)
  (and (Eapply? exp)
       (let ([expr1 (Eapply->expr1 exp)])
         (and (Eapply? expr1)
              (EOp? (Eapply->expr1 expr1))))))
(defprim (simple-exp? e)
  (or (simple-arg? e)
      (simple-apply? e)
      (simple-tuple? e)))
(defprim (simple-arg? e)
  (or (Elit? e) (EVar? e) (EFun? e) (Enil? e) (Ebool? e) (Econ? e) (EOp? e)))
(defprim (simple-apply? e)
  (and (Eapply? e)
       (or (and (simple-apply? (Eapply->expr1 e))
                (simple-arg? (Eapply->expr2 e)))
           (and (simple-fun? (Eapply->expr1 e))
                (simple-arg? (Eapply->expr2 e))))))
(defprim (simple-fun? e)
  (or (Econ? e) (EOp? e)))
(defprim (simple-tuple? e)
  (foldr (lambda (x y) (and x y)) '#t
         (map simple-arg? (Etuple->expr* e))))
;
;-------------------------------------------------------------------------
; Literal sort:
;
;  Lit -> (num Num) | (char Char) | (string Num*)
;
(defprim (LNum? lit)    (and (pair? lit) (equal? (car lit) 'num)))
(defprim (LChar? lit)   (and (pair? lit) (equal? (car lit) 'char)))
(defprim (LString? lit) (and (pair? lit) (equal? (car lit) 'string)))
(defprim 1 Lit->num cadr)
```

```
(defprim (Lit->char l) (integer->char (cadr l)))
(defprim-dynamic (Lit->String l) (LString (cdr l)))
;
(defprim-dynamic (LString s*)
  (if (emptys*? s*)
      (inVNil)
      (inV: (suspend (integer->char (s*->s1 s*)))
            (my-delay (LString (s*->sr s*))))))
(defprim 1 emptys*? null?)
(defprim 1 s*->s1 car)
(defprim 1 s*->sr cdr)
;
;-------------------------------------------------------------------------
; Script sorts
;
;   Scr   -> Def*
;   Def   -> (fundef Fun Eq*) | (tdef ... Cdef*)
;   Eq    -> (Pat* Alt* Def*)
;   Alt   -> (Expr Guard)
;   Guard -> Expr
;   Cdef  -> (con Atexpr*)
;
(defprim (fundef? def)
  (and (pair? def) (equal? (car def) 'fundef)))
(defprim (typedef? def)
  (and (pair? def) (equal? (car def) 'tdef)))
(defprim (conformal? def)
  (and (pair? def) (equal? (car def) 'conformal)))
(defprim 1 def*->def1 car)
(defprim 1 def*->def* cdr)
(defprim 1 def->fun cadr)
(defprim 1 def->eq* caddr)
(defprim 1 def->cdef* cadddr)
(defprim 1 cdef*->cdef1 car)
(defprim 1 cdef*->cdef* cdr)
(defprim 1 cdef->con car)
(defprim 1 cdef->atexpr* cadr)
;(defprim 1 conf->pat cadr)
;(defprim 1 conf->alt* caddr)
;(defprim 1 conf->def* cadddr)
(defprim 1 eq*->eq1 car)
(defprim 1 eq*->eq* cdr)
(defprim 1 eq->pat* car)
(defprim 1 eq->alt* cadr)
(defprim 1 eq->def* caddr)
(defprim 1 alt*->alt car)
(defprim 1 alt*->alt* cdr)
(defprim 1 alt->expr car)
(defprim 1 alt->guard cadr)
(defprim (inFundef name eq*) (list 'fundef name eq*))
(defprim (inEq pat* alt* def*) (list pat* alt* def*))
(defprim (inAlt e1 e2) (list e1 e2))
;
(defprim (funarity eq*)
  (foldr max 0
         (map (lambda (eq) (length (car eq))) eq*)))
(defprim (TrueGuard? g) (equal? g '(cons |True|)))
;
;-------------------------------------------------------------------------
; Pattern sort:
;
;  pat -> (pvar Var) | (plit Lit) | (ptuple Pat*)
;       | (pcon Con Pat*) | (ppair Pat Pat) | (pnil)
;
(defprim (pvar? pat)
  (and (pair? pat) (equal? (car pat) 'pvar)))
(defprim (plit? pat)
```

```
  (and (pair? pat) (equal? (car pat) 'plit)))
(defprim (ppair? pat)
  (and (pair? pat) (equal? (car pat) 'ppair)))
(defprim (pcon? pat)
  (and (pair? pat) (equal? (car pat) 'pcons)))
(defprim (ptuple? pat)
  (and (pair? pat) (equal? (car pat) 'ptuple)))
(defprim (pnil? pat) (equal? pat '(pnil)))
(defprim 1 pvar->var cadr)
(defprim 1 plit->lit cadr)
(defprim 1 ppair->p1 cadr)
(defprim 1 ppair->p2 caddr)
(defprim 1 pcon->con cadr)
(defprim 1 pcon->pat* caddr)
(defprim 1 ptuple->pat* cadr)
(defprim (ptuple->size pat) (length (cadr pat)))
(defprim 1 pat*->p1 car)
(defprim 1 pat*->pat* cdr)
(defprim (inP: pat1 pat2) (list 'ppair pat1 pat2))
(defprim (inPCon con pat*) (list 'pcons con pat*))
(defprim (inPTuple pat*) (list 'ptuple pat*))
;
; Special cases:
;
(defprim (pbool? pat)
  (or (equal? pat '(pcons |True| ()))
      (equal? pat '(pcons |False| ()))))
(defprim (pbool->bool pat)
  (equal? pat '(pcons |True| ())))
(defprim (pstring? pat)
  (and (plit? pat)
       (equal? (caadr pat) 'string)))
;
;---------------------------------------------------------------------------
;  Arithmetic Sequences sort:
;
(defprim (AS-e..? e)
  (and (ENone? (Eas->expr2 e))
       (ENone? (Eas->expr3 e))))
(defprim (AS-ee..? e)
  (and (not (ENone? (Eas->expr2 e)))
       (ENone? (Eas->expr3 e))))
(defprim (AS-e..e? e)
  (and (ENone? (Eas->expr2 e))
       (not (ENone? (Eas->expr3 e)))))
(defprim (ENone? exp) (equal? exp '(none)))
;
;---------------------------------------------------------------------------
; List comprehension sort:
;
;  Qual -> (qlist Pat Expr) | (qexp Expr)
;
(defprim (LCgen? q) (and (pair? q) (equal? (car q) 'qlist)))
(defprim 1 LCgen->pat caadr)
(defprim 1 LCgen->expr caddr)
(defprim 1 LCfilter->expr cadr)
;
;---------------------------------------------------------------------------
; Data sort:
;
(defprim-dynamic (VCon->con c) (vector-ref c 0))
(defprim-dynamic 2 Vcon-ref vector-ref)
(defprim-dynamic 2 VTuple-ref vector-ref)
(defprim-dynamic (inVcon con tuple)
  (let ([t1 (vector-copy tuple)])
    (vector-set! t1 0 con)
    t1))
```

```
(defprim-dynamic 1 inVCon0 vector)
(defprim-dynamic 2 inVCon1 vector)
(defprim-dynamic 3 inVCon2 vector)
(defprim-dynamic 4 inVCon3 vector)
(defprim-dynamic (vnil? v) (equal? v 'nil))
(defprim-dynamic (inVnil) 'nil)
(defprim-dynamic 1 V:? pair?)
(defprim-dynamic 2 inV: cons)
(defprim-dynamic (tupleToList a v)
  ((rec loop
        (lambda (n)
          (if (> n a)
              ()
              (cons (lambda () ((VTuple-ref (v) n))); = (my-delay (my-force ...))
                    (loop (add1 n))))))             ; This prevents v from being
   1))                                              ; forced until one of its
                                                    ; elements are needed.
(defprim-dynamic (listToTuple e*) (list->vector (cons (length e*) e*)))
(defprim-dynamic (apply-fun-d f vs)
  (if (null? vs)
      f
      (apply-fun-d (f (car vs)) (cdr vs))))
;
(defprim-dynamic 1 inVTuple0 vector)
(defprim-dynamic 2 inVTuple1 vector)
(defprim-dynamic 3 inVTuple2 vector)
(defprim-dynamic 4 inVTuple3 vector)
(defprim-dynamic (inVTuple* a e*) (list->vector (cons a e*)))
;
;-----------------------------------------------------------------------
; Description sort:
;
;   Ds -> (dlit Lit) | (dbool bool) | (dcon Con Ds*) | (dpair Ds Ds) | (dnil) |
;         (dtuple Ds*) | dbottom | (dnlit Lit*) | (dncon Con*)
;
(defprim (dlit? ds) (and (pair? ds) (equal? (car ds) 'Dlit)))
(defprim (dbool? ds) (and (pair? ds) (equal? (car ds) 'Dbool)))
(defprim (dcon? ds) (and (pair? ds) (equal? (car ds) 'Dcon)))
(defprim (dpair? ds) (and (pair? ds) (equal? (car ds) 'Dpair)))
(defprim (dtuple? ds) (and (pair? ds) (equal? (car ds) 'Dtuple)))
(defprim (dnil? ds) (and (pair? ds) (equal? (car ds) 'Dnil)))
(defprim (dnlit? ds) (and (pair? ds) (equal? (car ds) 'Dnlit)))
(defprim (dncon? ds) (and (pair? ds) (equal? (car ds) 'Dncon)))
(defprim (dbot? ds) (equal? ds 'Dbottom))
(defprim 1 dlit->lit cadr)
(defprim 1 dbool->bool cadr)
(defprim 1 dcon->con cadr)
(defprim 1 dcon->ds* caddr)
(defprim 1 dpair->d1 cadr)
(defprim 1 dpair->d2 caddr)
(defprim 1 dtuple->ds* cadr)
(defprim 1 dnlit->set cadr)
(defprim 1 dncon->set cadr)
(defprim (indcon con ds*) (list 'Dcon con ds*))
(defprim (indpair ds1 ds2) (list 'Dpair ds1 ds2))
(defprim (indlit lit) (list 'Dlit lit))
(defprim (indbool bool) (list 'Dbool bool))
(defprim (inDtuple l) (list 'DTuple l))
(defprim (newdnlit lit) (list 'Dnlit (list lit)))
(defprim (indnlit litset) (list 'Dnlit litset))
(defprim (newdcon con a) (list 'Dcon con (makeDbottomlist a)))
(defprim (newdpair) '(Dpair Dbottom Dbottom))
(defprim (newdnil) (list 'Dnil))
(defprim (NewDtuple n) (inDTuple (makedbottomlist n)))
(defprim (newdbottom) 'Dbottom)
(defprim (newdncon con ctab)
  (list 'Dncon (delete con (lookupconfamily con ctab))))
```

```
(defprim (adddnlit lit ds) (list 'DNlit (cons lit (DNlit->set ds))))
(defprim (deldncon con ds) (list 'DNcon (delete con (DNcon->set ds))))
(defprim (arity-con con ctab) (cadr (assoc con (join ctab))))
(defprim (ctab->con* ctab) (map car (join ctab)))
(defprim (unaryfamily con ctab) (= (length (lookupconfamily con ctab)) 1))
(defprim (makedbottomlist a)
  (if (zero? a)
      ()
      (cons 'Dbottom (makeDbottomlist (sub1 a)))))
(defprim (lookupconfamily con ctab)
  (if (null? ctab)
      ()
      (let ((fam (car ctab)))
        (if (assoc con fam)
            (map car fam)
            (lookupconfamily con (cdr ctab))))))
;
;-----------------------------------------------------------------------
; Occurrence sort:
;
;   Occ -> top | (Num Occ) | (head Occ) | (tail Occ) | free | (con Occ)
;
(defprim (OTop? occ) (equal? occ 'top))
(defprim (OConArg? occ) (and (pair? occ) (number? (car occ))))
(defprim (OTupleArg? occ) (and (pair? occ) (number? (car occ))))
(defprim (O:Head? occ) (and (pair? occ) (equal? (car occ) 'head)))
(defprim (O:Tail? occ) (and (pair? occ) (equal? (car occ) 'tail)))
(defprim (Free? occ) (equal? occ 'free))
(defprim 1 OConArg->arg car)
(defprim 1 OTupleArg->arg car)
(defprim (inOTop) 'top)
(defprim (inOArg n occ) (cons n occ))
(defprim (inOCon occ) occ)
(defprim (inOTuple occ) occ)
(defprim (inOHead occ) (cons 'head occ))
(defprim (inOTail occ) (cons 'tail occ))
(defprim (inOConName occ) (cons 'con occ))
;
;-----------------------------------------------------------------------
; Basic operators:
;
(defprim (basic-operators)
  '(!append !listdiff !times !uminus !plus !minus !and !or !not !equal
            !greater !notless !notequal !less !notgreater !divide !div
            !mod !power !length !index !compose strict))
(defprim (Binop? op)
  (or (Strictbinop? op) (Strictlazybinop? op) (Lazybinop? op)))
(defprim (Uop? op)
  (or (Strictuop? op) (Lazyuop? op)))
(defprim (Strictbinop? op)
  (member op '(!times !plus !minus !equal !greater !notless !notequal
               !less !notgreater !divide !div !mod !power !index strict)))
(defprim (Strictlazybinop? op)
  (member op '(!or !and)))
(defprim (Lazybinop? op)
  (member op '(!append !listdiff)))
(defprim (Strictuop? op)
  (member op '(!not !uminus)))
(defprim (Lazyuop? op)
  (member op '(!length)))
(defprim (Compose? op)
  (equal? op '!compose))
;
;-----------------------------------------------------------------------
; Evaluation of dynamic basis operations:
;
(defprim-dynamic (Benv op)
```

```
  (cond
; List operations:
   ((equal? op '!append)      (lazy-binop lazy-append))
   ((equal? op '!listdiff)    (lazy-binop lazy-list-subtract))
   ((equal? op '!length)      (lazy-unop lazy-length))
   ((equal? op '!index)       (strict-binop lazy-index))
; Logical operations:
   ((equal? op '!and)         (lazy-binop lazy-and))
   ((equal? op '!or)          (lazy-binop lazy-or))
   ((equal? op '!not)         (lazy-unop lazy-not))
; Relational operations:
   ((equal? op '!greater)     (strict-relop >))
   ((equal? op '!notless)     (strict-relop >=))
   ((equal? op '!equal)       (strict-relop struct-equal?))
   ((equal? op '!notequal)    (strict-relop not-equal?))
   ((equal? op '!notgreater)  (strict-relop <=))
   ((equal? op '!less)        (strict-relop <))
; Numerical operations:
   ((equal? op '!plus)        (strict-binop +))
   ((equal? op '!minus)       (strict-binop -))
   ((equal? op '!times)       (strict-binop *))
   ((equal? op '!divide)      (strict-binop /))
   ((equal? op '!div)         (lazy-binop lazy-div))
   ((equal? op '!mod)         (strict-binop modulo))
   ((equal? op '!uminus)      (lazy-unop lazy-uminus))
   ((equal? op '!power)       (lazy-binop lazy-power))
; Function composition:
   ((equal? op '!compose)     (lazy-binop lazy-composition))
   ((equal? op 'strict)       (strict-binop strict))
   (else (error '*** "Unknown basis-operator: ~s" op))))
;
;---------------------------------------------------------------------------
; Basic operations:
;
(defprim-dynamic (struct-equal? v1 v2)
  (cond
   ((and (vector? v1) (vector? v2))
    (vector-equal? v1 v2))
   ((and (pair? v1) (pair? v2))
    (and (struct-equal? (my-force (car v1)) (my-force (car v2)))
         (struct-equal? (my-force (cdr v1)) (my-force (cdr v2)))))
   ((equal? v1 'nil)
    (equal? v2 'nil))
   ((or (procedure? v1) (procedure? v2))
    #f)
   (else
    (equal? v1 v2))))

(defprim-dynamic (vector-equal? v1 v2)
  (and (equal? (vector-ref v1 0) (vector-ref v2 0))
       ((rec loop
             (lambda (n)
               (if (= n 0)
                   #t
                   (and (struct-equal?
                         (my-force (vector-ref v1 n))
                         (my-force (vector-ref v2 n)))
                        (loop (sub1 n))))))
        (sub1 (vector-length v2)))))

(defprim-dynamic (lazy-div v1 v2)
  (quotient (my-force v1) (my-force v2)))

(defprim-dynamic (lazy-uminus x)
  (- (my-force x)))

(defprim-dynamic (lazy-power x y)
```

```
  (power (my-force x) (my-force y)))

(defprim (power x y)
  (if (and (integer? x) (integer? y))
      (cond
       ((zero? y) 1)
       ((negative? y) (/ 1 (ipower x (- 0 y))))
       (else (ipower x y)))
      (exp (* y (log x)))))

(defprim (ipower x y)
  ((rec loop
        (lambda (y)
          (cond
           ((even? y)
            (let ((v (loop (/ y 2))))
              (* v v)))
           ((= 1 y) x)
           (else
            (* x (loop (sub1 y)))))))
   y))

(defprim-dynamic (not-equal? x y)
  (not (struct-equal? x y)))

(defprim-dynamic (lazy-not x)
  (not (my-force x)))

(defprim-dynamic (lazy-and x y)
  (and (my-force x) (my-force y)))

(defprim-dynamic (lazy-or x y)
  (or (my-force x) (my-force y)))

(defprim-dynamic (lazy-cons x y) (cons x y))

(defprim-dynamic (lazy-append e1 e2)
  ((rec loop
        (lambda (e1)
          (let ((v1 (my-force e1)))
            (if (equal? v1 'nil)
                (my-force e2)
                (cons (car v1)
                      (my-delay (loop (cdr v1))))))))
   e1))

(defprim-dynamic (lazy-list-subtract x y)
  ((rec loop
        (lambda (x y)
          (let ((vy (my-force y)))
            (if (equal? vy 'nil)
                (my-force x)
                (loop
                 (let ((b (car vy)))
                   ((rec loop1
                         (lambda (l)
                           (let ((vl (my-force l)))
                             (if (equal? vl 'nil)
                                 (suspend 'nil)
                                 (let ((l1 (car vl)))
                                   (if (struct-equal? (my-force b) (my-force l1))
                                       (cdr vl)
                                       (my-delay
                                        (cons l1 (loop1 (cdr vl))))))))))
                    x))
                 (cdr vy))))))
   x y))
```

```
(defprim-dynamic (lazy-length x)
  ((rec loop
        (lambda (x)
          (let ((v (my-force x)))
            (if (equal? v 'nil)
                0
                (add1 (loop (cdr v)))))))
   x))

(defprim-dynamic (lazy-index l n)
  ((rec loop
        (lambda (l n)
          (if (pair? l)
              (if (zero? n)
                  (my-force (car l))
                  (loop (my-force (cdr l)) (sub1 n)))
              (error '*** "Index out of range."))))
   l n))

(defprim-dynamic (lazy-composition f g)
  (lambda (x)
    ((my-force f) (my-delay ((my-force g) x)))))

(defprim-dynamic (strict f v)
  (f (lambda () v)))
;
;-----------------------------------------------------------------------
; Evaluation of boolaens:
;
(defprim (Bool e) (equal? e '(cons |True|)))
;
;-----------------------------------------------------------------------
; Primitives for constructing tables:
;
(defprim (make-ctab scr)
  (if (null? scr)
      '()
      (let* ((d1     (def*->def1 scr))
             (newscr (def*->def* scr)))
        (if (TypeDef? d1)
            (cons (map (lambda (cd) (list (car cd) (length (cadr cd))))
                       (Def->cdef* d1))
                  (make-ctab newscr))
            (make-ctab newscr)))))
;
(defprim (collect-functions scr)
  (if (null? scr)
      '()
      (let* ((d1     (def*->def1 scr))
             (newscr (def*->def* scr)))
        (if (FunDef? d1)
            (cons (Def->fun d1) (collect-functions newscr))
            (collect-functions newscr)))))
;
(defprim (collect-conformals scr)
  (if (null? scr)
      '()
      (let* ((d1     (def*->def1 scr))
             (newscr (def*->def* scr)))
        (if (and (FunDef? d1) (= (funArity (Def->eq* d1)) 0))
            (cons (Def->fun d1) (collect-conformals newscr))
            (collect-conformals newscr)))))
;
(defprim (collect-constructors scr)
  (if (null? scr)
      '()
```

```
      (let* ((d1     (def*->def1 scr))
             (newscr (def*->def* scr)))
        (if (TypeDef? d1)
            (append (map car (Def->cdef* d1))
                    (collect-constructors newscr))
            (collect-constructors newscr)))))
;
;=========================================================================
; ********************************************************************** ;
;=========================================================================
; Pre-transformations:
;
;   Does:
;     1. Translate conformals into functions.
;     2. Changes var tags into fun for functions and op for basic operators.
;
(defprim (elab-scr scr)
  (let ([def* (trans-conformals scr)])
    (elab-def* def* (append (collect-functions def*)
                            (library-functions)))))
(defprim (elab-def* def* funs)
  (map (lambda (def) (elab-def def funs)) def*))
(defprim (elab-def def funs)
  (if (FunDef? def)
      (inFunDef (def->fun def) (elab-eq* (def->eq* def) funs))
      def))
(defprim (elab-eq* eq* funs)
  (map (lambda (eq) (elab-eq eq funs)) eq*))
(defprim (elab-eq eq funs)
  (let* ([def* (eq->def* eq)]
         [pat* (eq->pat* eq)]
         [newfuns (union (setdiff funs (Defvarpat* pat*))
                         (collect-functions def*))]
         [newpat* (map transstringpat pat*)]
         [newalt* (elab-alt* (eq->alt* eq) newfuns)])
    (inEq newpat* newalt* (elab-def* def* newfuns))))
(defprim (elab-alt* alt* funs)
  (map (lambda (alt) (elab-alt alt funs)) alt*))
(defprim (elab-alt alt funs)
  (inAlt (elab-expr (alt->expr alt) funs) (elab-expr (alt->guard alt) funs)))
(defprim (elab-expr* expr* funs)
  (map (lambda (expr) (elab-expr expr funs)) expr*))
(defprim (elab-expr expr funs)
  (cond
   ((EVar? expr)   (let ((var (EVar->var expr)))
                     (cond
                      ((member var funs)
                       (inEFun var))
                      ((member var (basic-operators))
                       (inEOp var))
                      (else
                       expr))))
   ((E:? expr)     (inE: (elab-expr (E:->Expr1 expr) funs)
                         (elab-expr (E:->Expr2 expr) funs)))
   ((ETuple? expr) (inETuple (elab-expr* (ETuple->expr* expr) funs)))
   ((EApply? expr) (inEapply (elab-expr (EApply->Expr1 expr) funs)
                             (elab-expr (EApply->Expr2 expr) funs)))
   ((EAS? expr)    (inEAS (elab-expr (EAS->Expr1 expr) funs)
                          (elab-expr (EAS->Expr2 expr) funs)
                          (elab-expr (EAS->Expr3 expr) funs)))
   ((ELC? expr)    (elab-lc expr funs))
   (else           expr)))
;
(defprim (elab-lc lc funs)
  (let ((res (elab-qual* (Elc->qual* lc) funs (Elc->expr lc))))
    (inLC (car res) (cdr res))))
;
```

```
(defprim (elab-qual* qual* funs expr)
  (if (null? qual*)
      (cons (elab-expr expr funs) ())
      (let ((qual (car qual*)))
        (if (LCgen? qual)
            (let ((res (elab-qual* (cdr qual*)
                                   (setdiff funs (Defvarpat (LCgen->pat qual)))
                                   expr)))
              (cons (car res)
                    (cons (inGen (LCgen->pat qual) (elab-expr (LCgen->expr qual)
                                                              funs))
                          (cdr re))))
            (let ((res (elab-qual* (cdr qual*) funs expr)))
              (cons (car res)
                    (cons (infilter (elab-expr (LCfilter->expr qual) funs))
                          (cdr res))))))))
(defprim (inGen pat expr) (list 'qlist (list pat) expr))
(defprim (infilter expr) (list 'qexp expr))
;
;
;
(defprim (transstringpat pat)
   (cond
   ((and (PLit? pat) (pstring? pat)) (unfold-string-pat pat))
   ((PCon? pat)   (inPCon (Pcon->con pat) (map transstringpat (PCon->pat* pat))))
   ((PTuple? pat) (inPTuple (map transstringpat (PTuple->pat* pat))))
   ((PPair? pat)  (inP: (transstringpat (PPair->P1 pat))
                        (transstringpat (PPair->P2 pat))))
   (else          pat)))

(defprim (unfold-string-pat pat)
  (let ((cl (cdadr pat)))
    ((rec loop
          (lambda (cl)
            (if (null? cl)
                '(pnil)
                (list 'ppair
                      (list 'plit (list 'char (car cl)))
                      (loop (cdr cl))))))
     cl)))
;
;================================================================
;
;  For  conformal pat = rhs  generate code:
;
;    $p1 = rhs
;
;  and for all variables v in pat generate code
;
;    FunDef $sel_v pat = var $p1
;    Conformal v = $sel_v $p1
;
(defprim (trans-conformals def*)
  (tc-def* def* 0))

(defprim (tc-def* def* n)
  (if (null? def*)
      ()
      (let ((def (car def*)))
        (cond
         ((typedef? def) (cons def (tc-def* (cdr def*) n)))
         ((conformal? def)
          (append (tc-def def n) (tc-def* (cdr def*) (add1 n))))
         ((fundef? def)
          (let ((fun (def->fun def)))
            (cons (list 'fundef fun
                        (map (lambda (eq)
```

```
                              (list (transstringpat (eq->pat* eq))
                                    (eq->alt* eq)
                                    (trans-conformals (eq->def* eq))))
                            (def->eq* def)))
                  (tc-def* (cdr def*) n))))
         (else ; type alias
          (tc-def* (cdr def*) n))))))

(defprim (tc-def def n)
  (let ((pat   (conf->pat def))
        (alt*  (conf->alt* def))
        (def*  (trans-conformals (conf->def* def)))
        (cname (make-cname n)))
    (cons (infundef cname (list (inEq () alt* def*)))
          (tc-pat pat cname))))

(defprim (tc-pat pat cname)
  (let ((fv* (Defvarpat pat)))
    (tc-var* fv* pat cname)))

(defprim (tc-var* fv* pat cname)
  (if (null? fv*)
      ()
      (let* ((v (car fv*))
             (sel (make-sel v)))
        (cons
          (infundef v
                    (list (inEq () ; empty pattern list
                                (list (inAlt (inEApply (inEVar sel) (inEVar cname))
                                             (inECon '|True|)))
                                ())) ; empty definition list
          (cons
            (infundef sel
                      (list (inEq (list pat)
                                  (list (inAlt (inEVar v) (inECon '|True|)))
                                  ())) ; empty definition list
            (tc-var* (cdr fv*) pat cname))))))

(defprim (make-cname n)
  (string->symbol (string-append "conf_" (format "~s" n))))

(defprim (make-sel v)
  (string->symbol (string-append "sel_" (symbol->string v))))
;
;-------------------------------------------------------------------------
; Primitives to find free variables:
;
;
; Primitives used by the FV functions.
;
(defprim (U f1 f2)
  (lambda (fv)
    (let ((fv1 (f2 fv)))
      (append fv1 (f1 (append fv1 fv))))))
(defprim (|\| f1 f2)
  (lambda (fv)
    (f1 (append (f2 fv) fv))))
(defprim (elem var)
  (lambda (fv)
    (if (member var fv) '() (list var))))
(defprim (empty fv) '())
;
;-----------------------------------------------------------------
;
; Free variables:
;
;     FV = free variables
```

```
;     DV = defined variables
;
;   FV(def*) = (U FV(def)) \ DV(def*)
;   FV(def)  = FV(eq*)
;   FV(eq*)  = U FV(eq)
;   FV(eq)   = [(FV(alt*) \ DV(def*)) U FV(def*)] \ DV(pat*)
;   FV(alt*) = U FV(alt)
;   FV(alt)  = FV(exp) U FV(guard)
;   FV(exp)  = ...
;
;   DV(pat*) = U DV(pat)
;   DV(pat)  = ...
;   DV(def*) = U DV(def)
;   DV(def)  =  fun
;
(defprim (FVDef* k def*)
  (|\| (foldr U empty (map (lambda (def) (FVDef k def)) def*)) (DVDef* def*)))
(defprim (FVDef k def) (FVeq* k (def->eq* def)))
(defprim (FVeq* k eq*)
  (foldr U empty (map (lambda (eq) (FVeq k eq)) eq*)))
(defprim (FVeq k eq)
  (|\| (U (|\| (FValt* k (eq->alt* eq)) (DVdef* (eq->def* eq)))
          (FVdef* k (eq->def* eq)))
       (DVPat* (eq->pat* eq))))
(defprim (FValt* k alt*)
  (foldr U empty (map (lambda (alt) (FValt k alt)) alt*)))
(defprim (FValt k alt)
  (U (FVexp k (alt->expr alt)) (FVexp k (alt->guard alt))))
(defprim (FVexp* k exp*)
  (foldr U empty (map (lambda (expr) (FVexp k expr)) exp*)))
(defprim (FVexp k exp)
  (cond
   ((EVar? exp)   (if (equal? k 'var) (elem (EVar->var exp)) empty))
   ((EFun? exp)   (if (equal? k 'fun) (elem (EFun->fun exp)) empty))
   ((E:? exp)     (U (FVexp k (E:->expr1 exp))
                     (FVexp k (E:->expr2 exp))))
   ((ETuple? exp) (FVexp* k (ETuple->exp* exp)))
   ((EApply? exp) (U (FVexp k (EApply->expr1 exp))
                     (FVexp k (EApply->expr2 exp))))
   ((EAS? exp)    (U (FVexp k (EAS->expr1 exp))
                     (U (FVexp k (EAS->expr2 exp))
                        (FVexp k (EAS->expr3 exp)))))
   ((ELC? exp)    (FVlc k exp))
   (else          empty)))
(defprim (FVlc k lc)
  (U (|\| (FVexp k (Elc->expr lc)) (DVqual* (Elc->qual* lc)))
     (FVqual* k (reverse (Elc->qual* lc)))))
(defprim (FVqual* k qual*)
  (if (null? qual*)
      empty
      (let ((qual (car qual*)))
        (if (LCgen? qual)
            (U (|\| (FVexp k (LCgen->expr qual))
                    (DVqual* (cdr qual*)))
               (FVqual* k (cdr qual*)))
            (U (|\| (FVexp k (LCfilter->expr qual)) (DVqual* (cdr qual*)))
               (FVqual* k (cdr qual*))))))))
(defprim (DVpat* pat*) (foldr U empty (map DVpat pat*)))
(defprim (DVpat pat)
  (cond
   ((PVar? pat)   (elem (PVar->var pat)))
   ; ((PPlus? pat)  (elem (PPlus->var pat)))
   ((PCon? pat)   (foldr U empty (map DVpat (PCon->pat* pat))))
   ((PTuple? pat) (foldr U empty (map DVpat (PTuple->pat* pat))))
   ((PPair? pat)  (U (DVpat (PPair->P1 pat)) (DVpat (PPair->P2 pat))))
   (else          empty)))
(defprim (DVdef* def*) (foldr U empty (map DVdef def*)))
```

```
(defprim (DVdef def)
  (if (FunDef? def)
      (elem (Def->fun def))
      empty))
(defprim (DUFSdef* def*) (foldr U empty (map DUFSdef def*)))
(defprim (DUFSdef def)
  (if (and (FunDef? def) (> (Def->arity def) 0))
      (elem (Def->fun def))
      empty))
(defprim (DCFSdef* def*) (foldr U empty (map DCFSdef def*)))
(defprim (DCFSdef def)
  (if (and (FunDef? def) (zero? (Def->arity def)))
      (elem (Def->fun def))
      empty))
(defprim (DVqual* qual*) (foldr U empty (map DVqual qual*)))
(defprim (DVqual qual)
  (if (LCgen? qual) (DVpat (LCgen->pat qual)) empty))
;
(defprim (freefunsdef def)   ((FVdef 'fun def) '()))
(defprim (freevardef* def*)  ((FVdef* 'var def*) '()))
(defprim (freevardef def)    ((FVdef 'var def) '()))
(defprim (freevareq* eq*)    ((FVeq* 'var eq*) '()))
(defprim (freevaralt* alt*)  ((FValt* 'var alt*) '()))
(defprim (freevarexp exp fv) ((FVexp 'var exp) fv))
;
(defprim (Defvarpat* pat*)  ((DVpat* pat*) ()))
(defprim (Defvarpat pat) ((DVpat pat) ()))
(defprim (new-ufs def* pat* ufs)
  ((U (DUFSdef* def*)
     (|\| (foldr U empty (map elem ufs))
          (DVpat* pat*))) '()))
(defprim (new-cfs def* pat* cfs)
  ((U (DCFSdef* def*)
     (|\| (foldr U empty (map elem cfs))
          (DVpat* pat*))) '()))
;
;=============================================================================
; ************************************************************************** ;
;=============================================================================
; Miscellaneous:
;
(defprim (join l) (foldr append () l))
(defprim (delete e l)
  (join (map (lambda (x) (if (equal? x e) () (list x))) l)))
(defprim (union s1 s2)
  ((rec loop
        (lambda (s)
          (if (null? s)
              s2
              (let ((e (car s)))
                (if (member e s2)
                    (loop (cdr s))
                    (cons e (loop (cdr s))))))))
   s1))
(defprim (intersection s1 s2)
  ((rec loop
        (lambda (s)
          (if (null? s)
              ()
              (let ((e (car s)))
                (if (member e s2)
                    (cons e (loop (cdr s)))
                    (loop (cdr s)))))))
   s1))
(defprim (setdiff s1 s2)
  ((rec loop
        (lambda (s)
```

```
         (if (null? s)
             ()
             (let ((e (car s)))
               (if (member e s2)
                   (loop (cdr s))
                   (cons e (loop (cdr s))))))))
   s1))
(defprim (foldr o e l)
  (if (null? l)
      e
      (o (car l) (foldr o e (cdr l)))))
(defprim 2 / /)
(defprim 1 log log)
(defprim 1 round round)
(defprim 1 exp exp)
(defprim 1 integer? integer?)
;
;=============================================================================
; ************************************************************************** ;
;=============================================================================
; Standard constructor environment:
;
(defprim (standard-constructor cn)
  (if (equal? cn '!cons)
      (lambda (x) (lambda (y) (inV: x y)))
      (error '*** "Unknown constructor: ~s" cn)))
;
;-----------------------------------------------------------------------------
; Standard function environment:
;
(defprim (library-functions)
  '(converse map take drop hd tl foldr member filter concat iterate sqrt !wrap !show))

(defprim-dynamic (library-call n)
  (cond
   ((equal? n 'converse) (lambda (x) (lambda (y) (converse x y))))
   ((equal? n 'map)      (lambda (x) (lambda (y) (map-bawl x y))))
   ((equal? n 'take)     (lambda (x) (lambda (y) (take (my-force x) y))))
   ((equal? n 'drop)     (lambda (x) (lambda (y) (drop (my-force x) y))))
   ((equal? n 'hd)       (lambda (x) (hd x)))
   ((equal? n 'tl)       (lambda (x) (tl x)))
   ((equal? n 'foldr)    (lambda (x) (lambda (y) (lambda (z) (foldr-bawl x y z)))))
   ((equal? n 'member)   (lambda (x) (lambda (y) (member-bawl x y))))
   ((equal? n 'filter)   (lambda (x) (lambda (y) (filter x y))))
   ((equal? n 'concat)   (lambda (x) (concat x)))
   ((equal? n 'iterate)  (lambda (x) (lambda (y) (iterate x y))))
   ((equal? n 'sqrt)     (lambda (x) (sqrt-bawl (my-force x))))
   ((equal? n '!wrap)    (lambda (x)
                           (lambda (y)
                             (lambda (z) (lambda (q) (wrap x y z q))))))
   ((equal? n '!show)    (lambda (x) (show (my-force x))))
   (else
    (error '*** "Unknown function ~s" n))))

(defprim (converse f x y)
  (((my-force f) y) x))

(defprim (take n l)
  (if (integer? n)
      ((rec loop
            (lambda (n l)
              (if (positive? n)
                  (let ((vl (my-force l)))
                    (if (equal? vl 'nil)
                        'nil
                        (cons (car vl) (my-delay (loop (sub1 n) (cdr vl))))))
                  'nil)))
```

```
       n l)
      (error '*** "Take applied to fractional number.")))

(defprim (drop n l)
  (if (integer? n)
      ((rec loop
            (lambda (n l)
              (let ((fl (my-force l)))
                (if (positive? n)
                    (if (equal? fl 'nil)
                        'nil
                        (loop (sub1 n) (cdr fl)))
                    fl))))
       n l)
      (error '*** "Drop applied to fractional number.")))

(defprim (mapx f l)
  ((rec loop
        (lambda (l)
          (let ((fl (my-force l)))
            (if (equal? fl 'nil)
                'nil
                (cons (my-delay ((my-force f) (car fl)))
                      (my-delay (loop (cdr fl))))))))
   l))

(defprim (foldrx v_0 v_1 v_2)
  (let ((x (my-force v_2)))
    (if (vnil? x)
        (v_1)
        (let ((op (my-force v_0)))
          ((rec loop
                (lambda (x)
                  ((op (car x))
                   (save (lambda ()
                           (let ((x1 (my-force (cdr x))))
                             (if (vnil? x1)
                                 (v_1)
                                 (loop x1))))))))
           x)))))

(defprim (hd l)
  (let ((fl (my-force l)))
    (if (equal? fl 'nil)
        (error '*** "hd applied to empty list.")
        (my-force (car fl)))))

(defprim (tl l)
  (let ((fl (my-force l)))
    (if (equal? fl 'nil)
        (error '*** "tl applied to empty list.")
        (my-force (cdr fl)))))

(defprim (memberx l e)
  (if (vnil? l)
      ()
      (let ((e (e)))
        ((rec loop (lambda (l)
                     (if (struct-equal? e ((car l)))
                         #t
                         (let ((tl ((cdr l))))
                           (if (vnil? tl)
                               ()
                               (loop tl))))))
         l))))

(defprim (filter p l)
```

```
  (let ([fl (my-force l)])
    (if (equal? (my-force l) 'nil)
        'nil
        (let* ([fp (my-force p)]
               [fl (my-force l)]
               [l1 (car fl)])
          (if (fp l1)
              (inv: l1
                    (save
                     (lambda ()
                       ((rec loop
                             (lambda (l)
                               (if (equal? (my-force l) 'nil)
                                   'nil
                                   (let* ([fl (my-force l)]
                                          [l1 (car fl)])
                                     (if (fp l1)
                                         (let ([l2 (cdr fl)])
                                           (inv: l1
                                                 (save
                                                  (lambda ()
                                                    (loop l2)))))
                                         (loop (cdr fl)))))))
                        (cdr fl)))))
              ((rec loop
                    (lambda (l)
                      (if (equal? (my-force l) 'nil)
                          'nil
                          (let* ([fl (my-force l)]
                                 [l1 (car fl)])
                            (if (fp l1)
                                (let ([l2 (cdr fl)])
                                  (inv: l1
                                        (save
                                         (lambda ()
                                           (loop l2)))))
                                (loop (cdr fl)))))))
               (cdr fl)))))))

(defprim (concat l)
  ((rec loop
        (lambda (l)
          (let ((fl (my-force l)))
            (if (equal? fl 'nil)
                'nil
                (let ([l1 (cdr fl)])
                  (lazy-append
                   (car fl)
                   (save (lambda () (loop l1)))))))))
   l))

(defprim (iterate f a)
  ((rec loop
        (lambda (a)
          (inv: a
                (save (lambda ()
                        (loop (save (lambda () ((f) a))))))))))
   a))

(defprim 1 sqrtx sqrt)

(defprim (wrap inner rel left right)
  (if ((my-force inner) left)
      (((my-force rel) left) right)
      #f))

(defprim-dynamic (show v)
```

```
  (cond
   ((vector? v)
    (if (symbol? (vector-ref v 0))
        (lazy-append
         (my-delay
          (inV: (suspend (car (string->list "(")))
                (my-delay
                 (show-string (symbol->string (vector-ref v 0))))))
         (my-delay
          (lazy-append
           (my-delay
            (let ((a (sub1 (vector-length v))))
              (if (> a 0)
                  (lazy-append
                   (my-delay
                    (inV: (suspend (car (string->list " ")))
                          (my-delay (showseq v 1 a " "))))
                   (suspend 'nil)))))
           (my-delay
            (inv: (suspend (car (string->list ")")))
                  (suspend 'nil))))))
        (lazy-append
         (my-delay
          (inV: (suspend (car (string->list "(")))
                (my-delay
                 (let ((a (sub1 (vector-length v))))
                   (if (> a 0)
                       (showseq v 1 a ",")
                       (suspend 'nil))))))
         (my-delay
          (inV: (suspend (car (string->list ")")))
                (suspend 'nil))))))
   ((pair? v)
    (lazy-append
     (my-delay
      (inV: (suspend (car (string->list "[")))
            (my-delay
             (lazy-append
              (my-delay (show (my-force (car v))))
              (my-delay (showlistit (my-force (cdr v))))))))
     (my-delay
      (inV: (suspend (car (string->list "]")))
            (suspend 'nil)))))
   ((equal? v 'nil)
    (show-string "[]"))
   ((procedure? v)
    (show-string "<function>"))
   ((boolean? v)
    (if v (show-string "True") (show-string "False")))
   ((char? v)
    (show-string (string-append "'" (list->string (list v)) "'")))
   (else
    (show-string (format "~s" v)))))

(defprim-dynamic (showseq v n a sep)
  ((rec loop
        (lambda (n)
          (if (= n a)
              (show (my-force (vector-ref v n)))
              (lazy-append
               (my-delay (show (my-force (vector-ref v n))))
               (my-delay
                (inV: (suspend (car (string->list sep)))
                      (my-delay (loop (add1 n))))))))))
   n))

(defprim-dynamic (showlistit v)
```

```
  ((rec loop
        (lambda (v)
          (if (equal? v 'nil)
              'nil
              (inV: (suspend (car (string->list ",")))
                    (my-delay
                     (lazy-append
                      (my-delay (show (my-force (car v))))
                      (my-delay (loop (my-force (cdr v))))))))))
   v))

(defprim-dynamic (show-string v)
  (LString (map char->integer (string->list v))))
;
;=============================================================================
; ************************************************************************* ;
;=============================================================================
; Interpretation:
;
(defprim (interpret expr venv cenv fenv funs)
  (print-routine (E-D (elab-expr expr funs) venv cenv fenv)))

(defprim (debug-int expr scr)
  (let ((t1 (S scr)))
    (interpret expr
               (lambda (w) (error '*** "Unknown variable: ~s" w))
               (vector-ref t1 0)
               (vector-ref t1 1)
               (append (vector-ref t1 2) (library-functions)))))

(defprim (debug-run expr)
  (let ((t1 (S-0)))
    (interpret expr
               (init-venv)
               (vector-ref t1 0)
               (vector-ref t1 1)
               (append (vector-ref t1 2) (library-functions)))))

(defprim (load-scr file)
  (car (file->list file)))
;
;-----------------------------------------------------------------------------
; Evaluation of dynamic expressions:
;
(defprim (E-D expr venv cenv fenv)
  (cond
   ((ELit?   expr)     (L-D (ELit->lit expr)))
   ((EBool?  expr)     (Bool expr))
   ((EFun?   expr)     (fenv (EFun->fun expr)))
   ((ECon?   expr)     (cenv (ECon->con expr)))
   ((EOp?    expr)     (Benv (EOp->Op expr)))
   ((EVar?   expr)     (error '*** "Unknown variable: ~s" (EVar->var expr)))
   ((ENil?   expr)     (inVNil))
   ((E:?     expr)     (inV: (my-delay (E-D (E:->expr1 expr) venv cenv fenv))
                           (my-delay (E-D (E:->expr2 expr) venv cenv fenv))))
   ((ETuple? expr)     (listToTuple (E*-D (Etuple->expr* expr) venv cenv fenv)))
   ((EApply? expr)     ((E-D (EApply->expr1 expr) venv cenv fenv)
                        (my-delay (E-D (EApply->expr2 expr) venv cenv fenv))))
   ((EAS?    expr)     (AS-D expr venv cenv fenv))
   ((ELC?    expr)     (LC-D expr cenv fenv venv (lambda () (inVNil))))
   (else               (error '*** "Syntax error in expreesion: ~s" expr))))
;
(defprim (E*-D expr* venv cenv fenv)
  (if (null? expr*)
      '()
      (cons (my-delay (E-D (car expr*) venv cenv fenv))
            (E*-D (cdr expr*) venv cenv fenv))))
```

```
;
(defprim (L-D lit)
  (cond
   ((LNum? lit) (Lit->num lit))
   ((LChar? lit) (Lit->char lit))
   (else ; lit a string
    (lit->String lit))))
;
;-----------------------------------------------------------------------
; Arithmetic Sequences:
;
(defprim (AS-D expr venv cenv fenv)
  (let* ([expr1 (Eas->expr1 expr)]
         [expr2 (Eas->expr2 expr)]
         [expr3 (Eas->expr3 expr)])
    (cond
     ((AS-e..? expr)   ; expr has form [expr..]
      (from (E-D expr1 venv cenv fenv)))
     ((AS-ee..? expr)  ; expr has form  [expr,expr..]
      (let ([v1 (E-D expr1 venv cenv fenv)])
        (fromThen v1 (- (E-D expr2 venv cenv fenv) v1))))
     ((AS-e..e? expr)  ; expr has form [expr..expr]
      (fromTo (E-D expr1 venv cenv fenv) (E-D expr3 venv cenv fenv)))
     (else             ; expr has form [expr,expr..expr]
      (let* ([v1 (E-D expr1 venv cenv fenv)]
             [v3 (E-D expr3 venv cenv fenv)]
             [s (- (E expr2 venv cenv fenv) v1)])
        (if (>= s 0)
            (fromUpTo v1 s v3)
            (fromDownTo v1 s v3)))))))
;
;-----------------------------------------------------------------------
; List comprehension:
;
(defprim (LC-D elc cenv fenv venv vs)
  (let* ([expr0 (ELC->expr elc)]
         [qual* (ELC->qual* elc)])
    (if (null? qual*)
        (inV: (my-delay (E-D expr0 venv cenv fenv)) vs)
        (let* ([qual (car qual*)]
               [qual1* (cdr qual*)])
          (if (LCgen? qual)
              (G-D (LCgen->pat qual)
                 (lambda (venv1 v1)
                   (LC-D (inLC expr0 qual1*) cenv fenv venv1 v1))
                 venv vs (E-D (LCgen->expr qual) venv cenv fenv))
              (if (E-D (LCfilter->expr qual) venv cenv fenv)
                  (LC-D (inLC expr0 qual1*) cenv fenv venv vs)
                  (my-force vs)))))))
;
(defprim (G-D pat c venv vs1 vs)
  (if (vnil? vs)
      (my-force vs1)
      (P-D pat venv
         (lambda (venv1)
           (c venv1 (my-delay (G-D pat c venv vs1 (my-force (cdr vs))))))
         (lambda () (G-D pat c venv vs1 (my-force (cdr vs))))
         (car vs))))
;
;-----------------------------------------------------------------------
; Pattern matching :
;
(defprim (MP-D pat* venv sc fc vs)
  (if (null? pat*)
      (apply-fun-d (sc venv) vs)
      (P-D (pat*->p1 pat*) venv
         (lambda (venv1) (MP-D (pat*->pat* pat*) venv1 sc fc (cdr vs)))
```

```
        fc (car vs))))
;
(defprim (P-D pat venv sc fc v)
  (cond
   ((Plit? pat)
    (if (vequal? (my-force v) (L-D (Plit->lit pat))) (sc venv) (fc)))
   ((Pcon? pat)
    (let ([fv (my-force v)]
          [pat* (Pcon->pat* pat)])
      (if (equal? (VCon->con fv) (Pcon->con pat))
          (MP-D pat* venv sc fc (tupleToList (length pat*) (lambda () fv)))
          (fc))))
   ((Ppair? pat)
    (let ([fv (my-force v)])
      (if (pair? fv)
          (P-D (Ppair->p1 pat) venv
             (lambda (venv1) (P-D (Ppair->p2 pat) venv1 sc fc (cdr fv)))
             fc (car fv))
          (fc))))
   ((Pnil? pat)
    (if (vnil? (my-force v)) (sc venv) (fc)))
   ((PTuple? pat)
    (let ([pat* (PTuple->pat* pat)])
      (MP-D pat* venv sc fc (tupleToList (length pat*) v))))
   (else ; pat is a variable!
    (sc (upd (Pvar->var pat) v venv)))))
;
;-----------------------------------------------------------------------
; Function related to the evaluation of Arithmetic Sequences:
;
(defprim-dynamic (from a)
  ((rec loop
        (lambda (a)
          (cons (suspend a) (my-delay (loop (add1 a))))))
   a))
(defprim-dynamic (fromThen a d)
  ((rec loop
        (lambda (a)
          (cons (suspend a) (my-delay (loop (+ a d))))))
   a))
(defprim-dynamic (fromTo a limit)
  ((rec loop
        (lambda (a)
          (if (<= a limit)
              (cons (suspend a) (my-delay (loop (add1 a))))
              'nil)))
   a))
(defprim-dynamic (fromUpTo a d limit)
  ((rec loop
        (lambda (a)
          (if (<= a limit)
              (cons (suspend a) (my-delay (loop (+ a d))))
              'nil)))
   a))
(defprim-dynamic (fromDownTo a d limit)
  ((rec loop
        (lambda (a)
          (if (>= a limit)
              (cons (suspend a) (my-delay (loop (+ a d))))
              'nil)))
   a))
;
;=======================================================================
; ********************************************************************* ;
;=======================================================================
; Print routine:
;
```

```
;   Very representation dependent!!! => To be as fast as possible.
;
(defprim-dynamic (print-routine v)
  (cond
   ((vector? v)
    (if (symbol? (vector-ref v 0))
        (begin
          (display "(")
          (display (vector-ref v 0))
          (let ((a (sub1 (vector-length v))))
            (when (> a 0)
                  (display " ")
                  (flush-output-port)
                  (printseq v 1 a " ")))
          (display ")")
          (flush-output-port))
        (begin
          (display "(")
          (flush-output-port)
          (let ((a (sub1 (vector-length v))))
            (when (> a 0)
                  (printseq v 1 a ",")))
          (display ")")
          (flush-output-port))))
   ((pair? v)
    (display "[")
    (flush-output-port)
    (print-routine (my-force (car v)))
    (printlistit (my-force (cdr v)))
    (display "]")
    (flush-output-port))
   ((equal? v 'nil)
    (display "[]")
    (flush-output-port))
   ((procedure? v)
    (display "<function>")
    (flush-output-port))
   ((boolean? v)
    (display (if v "True" "False"))
    (flush-output-port))
   ((char? v)
    (display "'")
    (display v)
    (display "'")
    (flush-output-port))
   (else
    (display v)
    (flush-output-port))))

(defprim-dynamic (printseq v n a sep)
  ((rec loop
        (lambda (n)
          (if (= n a)
              (print-routine (my-force (vector-ref v n)))
              (begin
                (print-routine (my-force (vector-ref v n)))
                (display sep)
                (flush-output-port)
                (loop (add1 n))))))
   n))

(defprim-dynamic (printlistit v)
  ((rec loop
        (lambda (v)
          (if (equal? v 'nil)
              ()
              (begin
```

```
                (display ",")
                (flush-output-port)
                (print-routine (my-force (car v)))
                (loop (my-force (cdr v)))))))
   v))

;============================================================================
; ************************************************************************ ;
;============================================================================
;
; Save operations:
;
(defprim-dynamic (save s)
  (let ((v '())
(tag #f))
    (lambda ()
      (unless tag
      (set! v (s))
      (set! tag #t))
      v)))

(defprim-dynamic (save-at a s)
  (unless (cdr a)
  (set-car! a (s))
  (set-cdr! a #t))
  (car a))
;
; Conformal store:
;
(defprim-dynamic (make-cfst n)
  (list->vector
   ((rec loop
         (lambda (n)
           (if (zero? n)
               ()
               (cons (cons n '()) (loop (sub1 n))))))
    n)))
(defprim 2 conf-store-ref vector-ref)
(defprim (location name ncfs)
  (- (length ncfs) (length (member name ncfs))))
;----------------------------------------------------------------------
;
; Compile time error handling:
;
(defprim 3 compile-time-error error)
;
;----------------------------------------------------------------------------
; Update environments:
;
;   Static environment using association lists:
;
(defprim (upd-s u v r)
  (cons (cons u v) r))
;
(defprim (lookup-s u r)
  (let ((ap (assoc u r)))
    (if ap
        (cdr ap)
        'free)))

(defprim (init-voenv) '())

(defprim (find-new-fov fv voenv uo fo)
  (if (null? fv)
      '()
      (let* ((v1 (car fv))
             (occ (lookup-s v1 voenv)))
```

```
(if (and (member occ uo) (member occ fo))
    (cons v1 (find-new-fov (cdr fv) voenv uo fo))
    (find-new-fov (cdr fv) voenv uo fo)))))
```

Appendix F

Test programs

This appendix contains the programs used in the performance tests described in Chapter 16.

F.1 Miranda and BAWL Programs

```
|| The greatest common divisor function:
gcd a b = gcd (a-b) b, if a>b
        = gcd a (b-a), if a<b
        = a, if a=b

|| Solving  quadratic equations:
quadsolve a b c = [],                          if delta<0
                = [-b/(2*a)],                  if delta=0
                = [-b/(2*a) + radix/(2*a),
                   -b/(2*a) - radix/(2*a)], if delta>0
                  where
                    delta = b*b - 4*a*c
                    radix = sqrt delta

|| The factorial function:
fac0 0 = 1
fac0 n = n*fac0 (n-1)

fac1 n = product1 [1..n]

fac2 n = product2 [1..n]

|| The Ackerman's function:
ack 0 n = n+1
ack m 0 = ack (m-1) 1
ack m n = ack (m-1) (ack m (n-1))

|| Computing the n'th fibonacci number:
fib0 0 = 1
fib0 1 = 1
fib0 n = fib0 (n-1) + fib0 (n-2)

|| Adding the elements in a list:
sum1 [] = 0
sum1 (a:x) = a + sum1 x

|| Multiplying the elements in a list:
product1 [] = 1
product1 (a:x) = a * product1 x

|| Reverse a list:
reverse1 [] = []
reverse1 (a:x) = reverse1 x ++ [a]

|| A perfidious example:
```

```
answer = twice twice twice twice suc 0
twice f = f . f
suc = (+ 1)

foldrx op k [] = k
foldrx op k (a:x) = op a (foldrx op k x)

sum2 = foldrx (+) 0
product2 = foldrx (*) 1
and2 = foldrx (&) True
reverse2 = foldrx postfix []
           where
             postfix a x = x ++ [a]

|| The permutations of a list:
perms [] = [[]]
perms x  = [ a:y | a <- x; y <- perms (x--[a]) ]

factors n = [ i | i <- [1..n div 2]; n mod i = 0 ]

|| Hoare's "Quicksort" algorithm:
qsort [] = []
qsort (a:x) = qsort [ b | b <- x; b<=a ]
              ++ [a] ++
              qsort [ b | b <- x; b>a ]

|| The eight queens example:
queens 0 = [[]]
queens n = [ q:b | b <- queens (n-1); q <- [0..7]; safe q b ]
safe q b = and2 [ ~checks q b i | i <- [0..#b-1] ]
checks q b i = q=b!i  abs1 (q - b!i)=i+1
abs1 x = x, if x>0
       = -x, otherwise

|| Perfect numbers:
perfects = [ n | n <- [1..]; sum1 (factors n) = n ]
primes = sieve [ 2.. ]
         where
           sieve (p:x) = p : sieve [ n | n <- x; n mod p > 0 ]

|| Fibonacci's numbers:
fib1 0 = 1
fib1 1 = 1
fib1 n = flist1!(n-1) + flist1!(n-2)
flist1 = map fib1 [0..]

fib2 n = flist2!n
flist2 = 1:1:[flist2!n + flist2!(n+1)|n<-[0..]]

|| Hamming numbers:
hamming = 1 : merge (f 2) (merge (f 3) (f 5))
          where
            f a = [ n*a | n <- hamming ]
            merge (a:x) (b:y) = a : merge x (b:y), if a<b
                              = b : merge (a:x) y, if a>b
                              = a : merge x y,     otherwise

|| Example with user defined data structures:
tree ::= Nilt | Node num tree tree

t1 = foldrx f Nilt [1..10]
     where
      f x y = Node x y y

mirror Nilt = Nilt
mirror (Node a x y) = Node a (mirror y) (mirror x)
```

```
sumtree Nilt = 0
sumtree (Node a x y) = a + (sumtree x) + (sumtree y)

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
|| Tests:
gcdtest = map (gcd 28) [1..200]

quadsolvetest = sum1 (map (sum1.f) [2..402])
                where
                  f x = (quadsolve 1 x 1)

fac0test     = sum1 (map fac0 (take 2000 tens))
fac1test     = sum1 (map fac1 (take 2000 tens))
fac2test     = sum1 (map fac2 (take 2000 tens))
acktest      = ack 3 4
fib0test     = fib0 20
fib1test     = fib1 300
fib2test     = fib2 300
answertest   = answer
reverse1test = reverse1 (reverse1 (reverse1 (reverse1 [1..100])))
reverse2test = reverse2 (reverse2 (reverse2 (reverse2 [100,99..1])))
permstest    = #(perms [1..6])
factortest   = sum1 (concat (map factors [1..200]))
qsorttest1   = qsort [1..100]
qsorttest2   = qsort [100,99..1]
queenstest   = take 1 (queens 8)
perfectstest = take 3 perfects
primestest   = hd (drop 99 primes)
hammingtest  = hd (drop 1000 hamming)
treetest     = sumtree (mirror t1)
```

F.2 LML programs

```
-- gcdtest
let rec gcd a b = if a>b then gcd (a-b) b else
                  if a<b then gcd a (b-a)
                         else a
in
  map (gcd 28) [1..200]

-- quadsolvetest
let rec quadsolve a b c
  = let rec delta = b*.b -. 4.0*.a*.c
    and     radix = sqrt delta
    in
      if delta<0.0 then [] else
      if delta=0.0 then [0.0-.b/.(2.0*.a)]
                   else [0.0-.b/.(2.0*.a)+.radix/.(2.0*.a);
       0.0-.b/.(2.0*.a)-.radix/.(2.0*.a)]
and     sum1 []     = 0.0
    ||  sum1 (x . xs) = x +. (sum1 xs)
in
  sum1 (map sum1 [quadsolve 1.0 x 1.0 ;; x <- (map itof [2..402])])

-- fac0test
let rec fac0 n = if (n = 0#) then 1# else n *# fac0 (n -# 1#)
and     sum1 []     = 0#
    ||  sum1 (x . xs) = x +# (sum1 xs)
and     tens = 10# . tens
in
  sum1 (map fac0 (head 2000 tens))

-- fac1test
let rec fac1 n = product1 (map (stoI o itos) [1..n])
and     product1 [] = 1#
    ||  product1 (a . x) = a *# product1 x
```

```
and     sum1 []     = 0#
    ||  sum1 (x . xs) = x +# (sum1 xs)
and     tens = 10 . tens
in
  sum1 (map fac1 (head 2000 tens))

-- fac2test
let rec fac1 n = product1 (map (stoI o itos) [1..n])
and     product1 = foldrx (*#) 1#
and     foldrx op k [] = k
    ||    foldrx op k (a . x) = op a (foldrx op k x)
and     sum1 []     = 0#
    ||  sum1 (x . xs) = x +# (sum1 xs)
and     tens = 10 . tens
in
  sum1 (map fac1 (head 2000 tens))

-- acktest
let rec ack 0 n = n+1
    ||  ack m 0 = ack (m-1) 1
    ||  ack m n = ack (m-1) (ack m (n-1))
in
  ack 3 4

-- fib0test
let rec fib0 0 = 1
    ||  fib0 1 = 1
    ||  fib0 n = fib0 (n-1) + fib0 (n-2)
in
  fib0 20

-- fib1test
let rec fib1 0 = 1#
    ||  fib1 1 = 1#
    ||  fib1 n = flist1??(n-1) +# flist1??(n-2)
and     flist1 = map fib1 [0..]
in
  fib1 300

-- fib2test
let rec fib2 n = flist2??n
and     flist2 = 1# . 1# . [flist2??n +# flist2??(n+1) ;; n<-[0..]]
in
  fib2 300

-- reverse1test
let rec reverse1 []      = []
    ||  reverse1 (a . x) = reverse1 x @ [a]
in
  reverse1 (reverse1 (reverse1 (reverse1 [1..100])))

-- reverse2test
let rec reverse2 = foldrx postfixx []
                   where rec
                     postfixx a x = x @ [a]
and     foldrx op k [] = k
    ||  foldrx op k (a . x) = op a (foldrx op k x)
in
  reverse2 (reverse2 (reverse2 (reverse2 [1..100])))

-- factortest
let rec factors n = [ i ;; i <- [1..n / 2]; n and     sum1 []     = 0
    ||  sum1 (x . xs) = x + (sum1 xs)
in
  sum1 (conc (map factors [1..200]))

-- qsorttest1
```

```
let rec qsort [] = []
    ||  qsort (a . x) = qsort [ b ;; b <- x ; b<=a ]
              @ [a] @
              qsort [ b ;; b <- x ; b>a ]
in
  qsort [1..100]

-- qsorttest2
let rec qsort [] = []
    ||  qsort (a . x) = qsort [ b ;; b <- x ; b<=a ]
              @ [a] @
              qsort [ b ;; b <- x ; b>a ]
in
  qsort [100;99..1]

-- queenstest
let rec queens 0 = [[]]
    ||  queens n = [ q . b ;; b <- queens (n-1); q <- [0..7]; safe q b ]
and     safe q b = and2 [ ~(checks q b i) ;; i <- [0..length(b)-1] ]
and     checks q b i = q=b??i | abs1 (q - b??i)=i+1
and     abs1 x = if x>0 then x else -x
and     and2 = foldrx (&) true
and     foldrx op k [] = k
    ||  foldrx op k (a . x) = op a (foldrx op k x)
in
  head 1 (queens 8)

-- perfectstest
let rec factors n = [ i ;; i <- [1..n / 2]; n and     sum1 []     = 0
    ||  sum1 (x . xs) = x + (sum1 xs)
and     perfects = [ n ;; n <- [1..]; sum1 (factors n) = n ]
in
  head 3 perfects

-- hammingtest
let rec hamming
  = 1 . merge (f 2) (merge (f 3) (f 5))
    where rec f a = [ n*a ;; n <- hamming ]
    and   merge (a . x) (b . y)
            = if a<b then a . merge x (b . y) else
              if a>b then b . merge (a . x) y
                     else a . merge x y
in
  hd (tail 1000 hamming)

-- primestest
let rec primes = sieve (from 2)
and     sieve (p . ps) = p . sieve [x ;; x<-ps ; xin
  hd (tail 99 primes)

-- treetest
let rec type Tree = leaf + node Int Tree Tree
in
let rec foldrx f n [] = n
    ||  foldrx f n (x . xs) = f x (foldrx f n xs)
and     t1 = foldrx f leaf [1..12]
and     f x y = node x y y
and     mirror leaf = leaf
    ||  mirror (node i t1 t2) = node i (mirror t2) (mirror t1)
and     sumtree leaf = 0
    ||  sumtree (node i t1 t2) = sumtree t1 + i + sumtree t2
in
  sumtree (mirror t1)
```

Appendix G

Prisopgaven tekst

Da denne rapport er en basvarelse af Københavns Universitets prisopgave 1991 gengiver vi prisopgaves ordlyden her:

Optimering i implementation af højere ordens programmeringssprog. Der ønskes udviklet semantikbaserede algoritmer til effektiv implementation af funktionelle programmeringssprog, herunder sprog med højere ordens funktioner og doven evaluering. Blandt relevante emner er eksperimentel afprøvning af algoritmerne, effektivitetsevaluering af den producerede og korrekthedsbevis. Blandt relevante teknikker kan nævnes bindingstidstransformationer og udnyttelse af fælles delstrukturer i data og beregninger.

Bibliography

[Barendregt 1984] H. P. Barendregt, *The lambda calculus, its syntax and semantics,* North-Holland, 2. edition, 1984.

[Bird and Wadler 1988] Richard Bird and Philip Wadler, *Introduction to Functional Programming, Computer Science,* Prentice-Hall, 1988.

[Bondorf 1990a] Anders Bondorf, Automatic autoprojection of higher order recursive equations, in *ESOP'90, Copenhagen, Denmark. Lecture Notes in Computer Science 432,* edited by Neil D. Jones, pages 70–87, Springer-Verlag, May 1990.

[Bondorf 1990b] Anders Bondorf, *Self-Applicable Partial Evaluation,* PhD thesis, DIKU, University of Copenhagen, Denmark, 1990.

[Bondorf 1991a] Anders Bondorf, Automatic autoprojection of higher order recursive equations, *Science of Computer Programming* 17 (Selected papers of ESOP '90, the 3rd European Symposium on Programming),1-3 (December 1991) 3–34.

[Bondorf 1991b] Anders Bondorf, Compiling laziness by partial evaluation, in [Jones *et al.* 1991], pages 9–22, 1991.

[Bondorf 1991c] Anders Bondorf, Similix Manual, system version 4.0, Included in Similix distribution, September 1991.

[Bondorf 1992] Anders Bondorf, Improving Binding Times without Explicit CPS-Conversion, 1992. Forthcoming.

[Bondorf and Danvy 1991] Anders Bondorf and Olivier Danvy, Automatic autoprojection of recursive equations with global variables and abstract data types, *Science of Computer Programming* 16 (1991) 151–195.

[Bondorf and Mogensen 1990] Anders Bondorf and Torben Æ. Mogensen, Logimix: A Self-applicable Partial Evaluator for Prolog, Unpublished, 1990.

[Bondorf *et al.* 1990] Anders Bondorf, Neil D. Jones, Torben Æ. Mogensen, and Peter Sestoft, Binding time analysis and the taming of self-application, Unpublished, 1990.

[Christiansen and Jones 1982] H. Christiansen and N. D. Jones, *Mathematical foundation of a semantics-directed compiler generator,* Technical Report, DIAMI, Aarhus Universitet, 1982.

[Consel 1988] Charles Consel, New insights into partial evaluation: the SCHISM experiment, in *ESOP'88, Nancy, France. Lecture Notes in Computer Science 300,* edited by Harald Ganzinger, pages 236–247, Springer-Verlag, March 1988.

[Consel and Danvy 1989] Charles Consel and Olivier Danvy, Partial evaluation of pattern matching in strings, *Information Processing Letters* 30,2 (1989) 79–86.

[Consel and Danvy 1991a] Charles Consel and Olivier Danvy, For a better support of static data flow, in *Conference on Functional Programming and Computer Architecture, Cambridge, Massachusetts,* ACM Press, August 1991. To appear.

[Consel and Danvy 1991b] Charles Consel and Olivier Danvy, Static and Dynamic Semantics Processing, in *Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida,* pages 14–24, ACM, January 1991.

[Damas and Milner 1982] Luis Damas and Robin Milner, Principal type-schemes for functional programs, in *Nineth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico,* pages 207–212, 1982.

[Danvy 1991] Olivier Danvy, Semantics-directed compilation of nonlinear patterns, *Information Processing Letters* 37,6 (1991) 315–322.

[Dybkjær 1985] Hans Dybkjær, *Parsers and partial evaluation: an experiment,* Student Report 85-7-15, DIKU, University of Copenhagen, Denmark, July 1985.

[Dybkjær 1991] Hans Dybkjær, *Category Theory, Types, and Programming Languages,* PhD thesis, DIKU, University of Copenhagen, Denmark, March 1991.

[Dybvig 1987] R. Kent Dybvig, *The SCHEME Programming Language,* Prentice-Hall, New Jersey, 1987.

[Futamura and Nogi 1988] Yoshihiko Futamura and Kenroku Nogi, Genreralized Partial Computation, in *Partial Evaluation and Mixed Computation,* edited by Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, pages 133–151, North-Holland, 1988.

[Gomard 1989] Carsten K. Gomard, *Higher Order Partial Evaluation – HOPE for the Lambda Calculus,* Master's thesis, DIKU, University of Copenhagen, Denmark, student report 89-9-11, September 1989.

[Gomard 1991] Carsten K. Gomard, *Program Analysis Matters,* PhD thesis, DIKU, University of Copenhagen, Denmark, 1991.

[Gomard and Jones 1989] Carsten K. Gomard and Neil D. Jones, Compiler generation by partial evaluation: a case study, in *Proceedings of the Twelfth IFIP World Computer Congress,* 1989.

[Henglein 1991] F. Henglein, Efficient Type Inference for Higher-Order Binding-Time Analysis, Feb. 1991. submitted for publication.

[Holst and Gomard 1991] Carsten Kehler Holst and Carsten K. Gomard, Partial evaluation is fuller laziness, in *Symposium on Partial Evaluation and Semantics-Based Program Manipulation, Yale University, New Haven, Connecticut. SIGPLAN Notices, volume 26, 9,* pages 223–233, ACM Press, June 1991.

[Holst and Hughes 1991] Carsten Kehler Holst and John Hughes, Towards binding-time improvement for free, in [Jones *et al.* 1991], pages 83–100, 1991.

[Hudak and Wadler 1990] Paul Hudak and Philip Wadler, *Report on the programming language Haskell,* Technical Report, Yale University and Glasgow University, April 1990.

[Johnson 1975] S. C. Johnson, *YACC: Yet another compiler compiler.,* Technical Report 32, Bell laboratories, Murray Hill, New Jersey, 1975.

[Jones *et al.* 1985] Neil D. Jones, Peter Sestoft, and Harald Søndergaard, An experiment in partial evaluation: the generation of a compiler generator, in *Rewriting Techniques and Applications, Dijon, France. Lecture Notes in Computer Science 202,* edited by J.-P. Jouannaud, pages 124–140, Springer-Verlag, 1985.

[Jones *et al.* 1991] Simon L. Peyton Jones, Graham Hutton, and Carsten Kehler Holst, editors, *Functional Programming, Glasgow 1990. Workshops in Computing,* Springer-Verlag, August 1991.

[Jørgensen 1990] Jesper Jørgensen, *Developing a pattern matching compiler using partial evaluation,* student report 90-1-5, DIKU, University of Copenhagen, Denmark, January 1990.

[Jørgensen 1991] Jesper Jørgensen, Generating a Pattern Matching Compiler by Partial Evaluation, in [Jones *et al.* 1991], pages 177–195, 1991.

[Jørgensen 1992] Jesper Jørgensen, Generating a Compiler for a Lazy Language by Partial Evaluation., in *Nineteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico,* pages ?–?, January 1992. Accepted to appear.

[Landin 1965] P.J. Landin, A Correspondence Between ALGOL 60 and Church's Lambda-Notation: Pert 1, *Communications of the ACM* 8,2 (1965) 89–101.

[Launchbury 1991] J. Launchbury, *Projection Factorisations in Partial Evaluation, Distinguished Dissertations in Computer Science,* Cambridge University Press, 1991.

[Lesk and Schmidt 1975] M. Lesk and E. Schmidt, *Lex - a lexical analyzer generator.,* Technical Report 39, Bell laboratories, Murray Hill, New Jersey, 1975.

[Milner *et al.* 1990] Robin Milner, Mads Tofte, and Robert Harper, *The definition of Standard ML.,* MIT Press, 1990.

[Mogensen 1986] T. Mogensen, *The Application of Partial Evaluation to Ray-Tracing,* Master's thesis, DIKU, University of Copenhagen, Denmark, 1986.

[Mosses 1975] Peter Mosses, *Mathematical Semantics and Compiler Generation,* PhD thesis, Oxford University, 1975.

[Mosses 1979] Peter Mosses, *SIS-semantics implementation system, reference manual and user guide,* Technical Report, DIAMI Aarhus Universitet, 1979.

[Paulson 1982] L. Paulson, A Semantics-Directed Compiler generator, in *Ninth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. Albuquerque, New Mexico,* pages 224–233, 1982.

[Paulson 1984] L. Paulson, Compiler generation from denotational semantics, in *Methods and Tools for Compiler Construction,* edited by B. Lorho, pages 219–250, 1984.

[Peyton Jones 1987] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages, Computer Science,* Prentice-Hall, 1987.

[Rees and Clinger 1986] Jonathan Rees and William Clinger, Revised Report[3] on the Algorithmic Language Scheme, *Sigplan Notices* 21,12 (December 1986) 37–79.

[Romanenko 1988] Sergei A. Romanenko, A compiler generator produced by a self-applicable specialiser can have a surprisingly natural and understandable structure, in *Partial Evaluation and Mixed Computation,* edited by Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, pages 445–463, North-Holland, 1988.

[Rothwell 1991] Nick Rothwell, Functional Compilation from the Standard ML Core Language to Lambda Calculus, in *Fourth Annula Workshop on Functional Programming,* pages 494–524, 1991. Draft Proceedings.

[Schmidt 1986] David A. Schmidt, *Denotational Semantics, a Methodology for Language Development,* Allyn and Bacon, Boston, 1986.

[Sestoft 1985] Peter Sestoft, The structure of a self-applicable partial evaluator, in *Programs as Data Objects, Copenhagen, Denmark. Lecture Notes in Computer Science 217,* edited by Harald Ganzinger and Neil D. Jones, pages 236–256, Springer-Verlag, October 1985.

[Søndergaard 1989] Harald Søndergaard, *Semantics-Based Analysis and Transfromation of Logic Programs,* PhD thesis, DIKU, University of Copenhagen, Denmark, 1989.

[Turner 1982] David Turner, Recursion equations as a programming language., in *Functional Programming and Its Applications.,* edited by Darlington et al., pages 1–28, Cambridge University Press, 1982.

[Turner 1986] David Turner, An Overview of Miranda, *Sigplan Notices* 21,12 (December 1986) 158–166.

[Turner 1989] David Turner, Miranda Manual, version 2, On-line manual of the Miranda system, 1989.

[Vickers 1986] T. N. Vickers, *Translator Generation using Denotational Semantics,* PhD thesis, University of New South Wales, Australia, 1986.

[Wadler 1985] Philip Wadler, *Introduction to Orwell,* Technical Report, Programming Research Group, University of Oxford, 1985.

[Weis 1987] P. Weis, *Métacompilation très efficace à l'aide d'Opérateurs Sémantiques,* PhD thesis, INRIA, l'universite Paris VII, 1987.