Partial Evaluation of Standard ML

Master's Thesis

Lars Birkedal (birkedal@diku.dk)

Morten Welinder (terra@diku.dk)

DIKU
Department of Computer Science
University of Copenhagen
Universitetsparken 1
DK-2100 Copenhagen Ø
Denmark

Revised version of October 22, 1993

Abstract

This thesis describes offline partial evaluation of the core of Standard ML, a large typed functional language. Unlike previous partial evaluators for larger languages (like for instance Similix for a subset of Scheme or C-Mix for a subset of C) we have chosen not do to the partial evaluation directly, but to use an untraditional method which we call *the cogen approach* to transform a program into its generating extension. We show in this thesis that this approach is in many aspects superior to the traditional approach and that it eliminates the need for self-applying the specializer.

We develop a binding-time analysis based on non-standard type inference and produce a very efficient implementation of it using constraints. While this has been done before, we have for the first time succeeded in using the typedness of the source language to make the analysis simple and therefore more trustworthy.

To our best knowledge this thesis also describes the first successful strategy for partially evaluating complicated patterns with variable bindings. Earlier attempts have either been for a much simpler class of patterns or have stranded on the need/wish for self-application of the specializer. Note that partially evaluating complicated patterns is different from partially evaluating programs dealing with complicated patterns; that has been done successfully.

A complete system for partial evaluation of Standard ML with parsing, type checking, binding-time analysis, compiler generation, and pretty printing has been implemented and we report on some experiments with this system. We have not implemented everything described in the thesis, though. Details can be found in Chapter 6.

Preface

This thesis is submitted in partial fulfillment of the requirements for a Danish Master's Degree (Candidatus Scientiarum) for both authors. It contains work done from August 1992 to August 1993. Our advisor on the project has been Professor Neil D. Jones at DIKU, the Department of Computer Science at the University of Copenhagen.

All work reported in this thesis has been made solely by the authors. Most of the work is based on the work of others, though, and we have made our best efforts to provide references to our sources.

Thesis

The following thesis has been the background of this project. It was inspired by ideas presented to us by Torben Mogensen, DIKU. The underlying principles were developed by [BHOS76] and later [Hol89] but have been neglected since then.

> *Partial evaluation by means of a hand-written compiler generator is 1) possible in practice, 2) simpler than traditional partial evaluation especially when it comes to strongly typed languages, and 3) better suited for attacking some of the harder problems of partial evaluation like pattern matching.*

Outline

The structure of our thesis is as follows: after introducing the generating extension approach to partial evaluation we define the language that we treat in terms of Standard ML.

In Chapter 3 we then discuss specialization of the different constructs in the language, i.e., what the generating extensions produced should look like.

Based on this analysis we then in Chapter 4 formally specify the process. This is done by introducing a two-level language on which we impose a non-standard type discipline and semantics that produces generating extensions.

A very efficient binding-time analysis for producing correct two-level versions from one-level programs is then developed in Chapter 5. The algorithm is based on [Hen91] but we have made some corrections, enhancements, and major simplifications. Even though the analysis is one of the strongest developed until now (in the sense that it handles a very complicated language) it is one of the simplest. This was possible by using available type information.

We have implemented most of the described algorithms and in Chapter 6 we report on our implementation and experiments with it.

In Chapter 7 we finally compare our work with related work of others and make our conclusions.

For Danish readers we supply a summary in Chapter 8.

Reader's Prerequisites

To obtain the most of this text the reader should have some general theoretical knowledge about partial evaluation. We do reformulate the basics because we follow a different approach but we do not elaborate. We expect a level roughly corresponding to [JSS89] or [JGS93, Chapters 4 and 5].

As we deal with Standard ML we also expect the reader to know Standard ML. We will explain the finer details like the generativity of exceptions so knowledge corresponding to [Pau91] is sufficient. Since we base our treatment of Standard ML on its Bare Language we expect the reader to have access to [MTH90] that defines the language and [MT91] that comments the definition. We will refer the reader to these two books when needed. Readers without knowledge of Standard ML but with experience with other typed functional languages and some imagination will very likely also obtain some benefit from reading this thesis.

Reader's Guide

The thesis is meant to be read from A to Z, but readers with prior experience in the field may choose different. We warn, however, that Chapter 4 depends heavily on Chapter 3.

Large parts of the chapter on binding-time analysis (Chapter 5) is self-contained. Readers confident with partial evaluation may start there. The section on experiments (Section 6.2) with appendices is also fairly self-contained.

As a special service for Danish readers Chapter 8 is a summary in Danish. We encourage Danish readers to start there, and then to go on with Chapter 1.

For the reader's convenience we have placed the bibliography and an index at the very end of the thesis.

Preface to the Revision

This thesis was revised slightly in October 1993 on the basis of comments from many readers. Mainly the revision fixes a large number of spelling and grammar errors; it does, however, also improve on the exposition and corrects a couple of genuine errors of thought. New and exciting errors have been introduced instead of those corrected.

Acknowledgements

We wish to thank *Neil D. Jones* for supervising this project and for providing contacts and pointers. Overview is a rare quality in the rapidly growing world of partial evaluation.

Special thanks go to *Lars Ole Andersen* for always useful comments on coding problems, for proof-reading this thesis, and for useful ideas on presentation and goals of the project. We believe that the fact that Lars only recently finished his own master's thesis made his advice "down-to-earth" and thus especially useful.

Thanks to *Anders Bondorf* and *Jesper Jørgensen* for discussions on different aspects of Similix and for pointing out several pitfalls before or after we stepped into them. When we did fall into the holes it was most often because we did not use the available help!

Thanks also go to *Torben Mogensen* for advertising the cogen-idea and for useful hints on data types, to *Fritz Henglein* for discussions on binding-time analysis, and to *Mads Tofte* for his deep knowledge of Standard ML.

We also wish to thank the entire Topps Group (aka "The Silo of Semantics") at DIKU as a whole for providing the right environment for creating, researching, and evaluating new ideas. The group is well-functioning and always willing to help!

Finally we wish to thank all the people who helped us proof-reading this work. The list is too numereous to show here, but you know who you are.

Contents

Chapter 1

Introduction

> *Next to Frodo on his right sat a dwarf of important appearance, richly dressed. His beard, very long and forked, was white, nearly as white as the snow-white cloth of his garments. He wore a silver belt, and round his neck hung a chain of silver and diamonds. Frodo stopped eating to look at him.*
>
> *'Welcome and well met!' said the dwarf, turning towards him. Then he actually rose from his seat and bowed. 'Glóin at your service,' he said, and bowed still lower.*
>
> — J. R. R. TOLKIEN, *Lord of the Rings (1954)*

A large class of similar computational problems can be solved in essentially two different ways: either by a specific program for each problem or by a general, parameterized program solving all the problems. A specific program is almost always more efficient than the general program, but the general program tends to be easier to write, maintain, and, moreover, given a new problem one does not have to write a new specific program but instead one can employ the general program.

In this thesis we study the problem of automatically turning a general, parameterized program into an efficient specialized program applicable to one given problem. This way we can obtain the best of two worlds, efficiency and generality. We study the problem in the context of Standard ML.

1.1 Computation in One Stage Or More

Turning a general program into a specialized program can be conceived of as turning a one-stage program into a two-stage program. For instance, an interpreter for a language L is a general program which is parameterized on an L program and the input to the L program. The interpreter is a one-stage program as the interpretation is performed in one step. A compiler on the other hand is a two-stage program. It takes the L program as input and delivers a target program, which afterwards can be executed on the input and give the result. The target program can be conceived of as a specialized version of the interpreter in the sense that it can only be used for problems specified by the L program.

Moreover, it is well-known that it is much more efficient to execute a compiled program than interpreting the original program, and that it is easier to write an interpreter than it is to write a compiler. We will see that the methods developed in this thesis are sufficient to *automatically* turn a one-stage program like an interpreter into a two-stage program like a compiler.

1.1.1 Generating Program Generators

The compiler from above is a program generator as it returns programs as results. So what we are looking for is a program, which we will call `cogen` for historical reasons, that generates program generators. To simplify matters we will without loss of generality assume that the general programs we want to specialize all take two arguments. The first one which describes the particular problem we will call static, the other we will call dynamic. The situation can then be pictured as in Figure 1, which is borrowed from [JGS93]. The `cogen` program accepts a general program `p`, and turns it into a

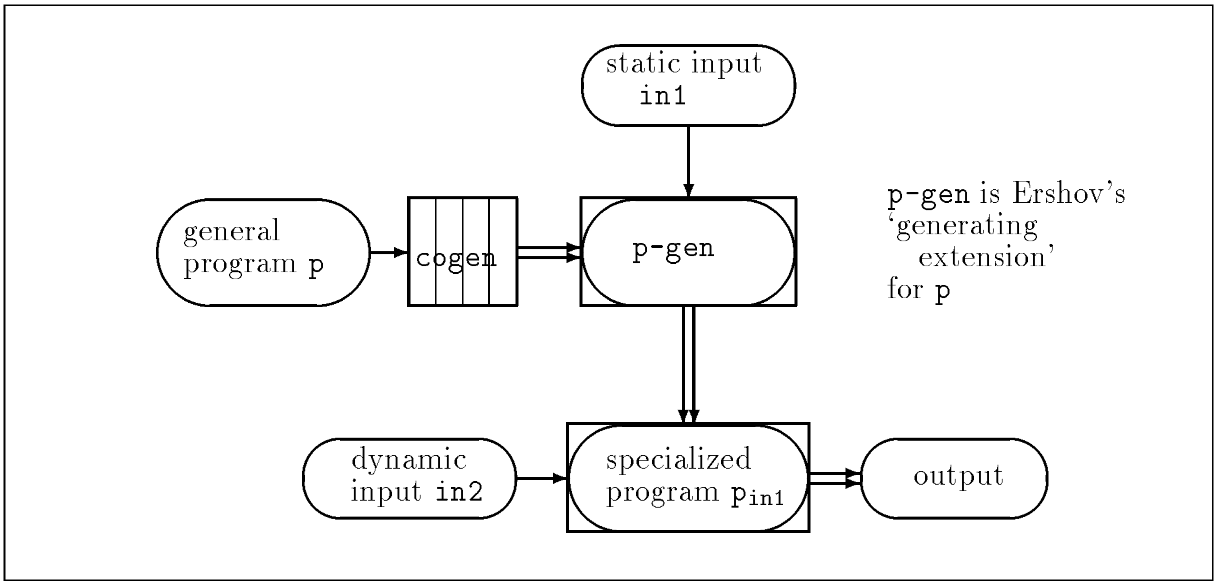

Figure 1: A Generator of Program Generators

program generator, `p-gen`. When `p-gen` is applied to the static argument of the general program, it produces a specialized program. This specialized program yields, when applied to the dynamic argument, a result which is equal to the result `p` would return when applied to both the static and dynamic argument. Notice how the one-stage program `p` has been turned into a two-stage program `p-gen`. The two-stage program `p-gen` is called the *generating extension* of `p`.

Jones, Gomard, and Sestoft [JGS93, Page 11] remark that

> *It would be wonderful to have a program generator generator, but it is far from clear how to construct one.*

In this thesis we actually develop such a program generator generator. Traditionally, the `cogen` and `p-gen` boxes in Figure 1 have been merged into one box, called `mix`, which takes both the general program and the static argument as arguments and returns a specialized program. The reason has been to simplify matters. In fact, it is possible to obtain the program generator generator indirectly by self-application of `mix`. A natural question then is: why consider developing a program generator generator directly if it is simpler to write `mix`? The reason is that there are certain problems with the program generator generators obtained by self-application, in particular when it comes to specializing strongly typed programs. We discuss these problems and how they can be solved by writing a program generator generator directly in the following sections.

1.2 Traditional Partial Evaluation

The `mix` program from above is called a *partial evaluator* or a *program specializer*. A partial evaluator can be characterized by whether it is *on-line* or *off-line*.

An on-line partial evaluator decides on the fly which operations can be performed at specialization time (i.e., when `mix` is executed), and which must be deferred to runtime. The partial evaluators [Mey91], [MS92], and [WCRS91] are all on-line. In off-line partial evaluators it is decided *a priori* to the specialization proper which operations to perform at which time. This is done by a so-called *binding-time analysis*. The partial evaluators [Bon90], [GJ89], [GJ91], [And92], and [Lau91] are all off-line and self-applicable.

It has been argued [BJMS88] that off-line specialization is superior when the goal is self-application of the specializer. To the best of our knowledge it is in fact the case that all successfully implemented self-applicable partial evaluators have been off-line. In this section we shall only be concerned with self-applicable partial evaluation as our goal as remarked above is to be able to obtain a program generator generator. Moreover, we shall be concerned with partial evaluation of strongly typed languages, as Standard ML is the prime example of a strongly typed language.

Notation

We need some notation in order to distinguish between program texts, values, program representations, and program meanings as we shall deal with all four concepts.

For any program text, `p`, written in language L we let $[\![\mathtt{p}]\!]_L$ denote the meaning of the program according to the semantics of the language L. Alternatively we may say that $[\![\cdot]\!]_L$ is an interpreter (as an abstract, mathematical function) for the language L. We will often omit the subscript when there is no chance for misunderstandings.

An interpreter written in a strongly typed language must represent the values of the language it interprets in some uniform data type Val, as the values can have different types. For instance, in the ML Kit [BRTT93] which is an interpreter for full Standard ML, a data type closely corresponding to the semantic object Val used in the Definition of Standard ML [MTH90] is used to represent SML values. We consider self-application so we assume that the interpreter is written in the language it interprets. When `v` is a value of some type we let $\overline{\mathtt{v}}^{\mathrm{Val}}$ be the encoding of `v` into the value data type. Likewise, let $\overline{\mathtt{p}}^{\mathrm{Pgm}}$ be the encoding of a program `p` into a data type Pgm. We will assume that the function $\mathtt{p} \mapsto \overline{\mathtt{p}}^{\mathrm{Pgm}}$ is injective, i.e., that we can remove the representation when we need to.

Program Specialization of Strongly-Typed Languages

We now reformulate the so-called Futamura projections in the setting of a strongly typed language in order to explain the above mentioned problems. This has also been done by Andersen [And92] and Launchbury [Lau91]. As a self-applicable partial evaluator can be conceived of as a smart self-interpreter [JGS93], we first consider applying a self-interpreter to itself to simplify the presentation.

Assume we apply a self-interpreter, `sint`, to itself, i.e., we apply the self-interpreter to the self-interpreter, a program `p` and some input data `d` to that program. The interpreted self-interpreter must then be represented in the Pgm data type; the program `p` must be represented as a Pgm value as input to the interpreted self-interpreter but subsequently be encoded as a value of Val data type as it is input to the running self-interpreter. Likewise for the data. Using the notation from above this is expressed as

$$[\![\mathtt{sint}]\!]\left(\overline{\mathtt{sint}}^{\mathrm{Pgm}}, \overline{\overline{\mathtt{p}}^{\mathrm{Pgm}}}^{\mathrm{Val}}, \overline{\overline{\mathtt{d}}^{\mathrm{Val}}}^{\mathrm{Val}}\right)$$

Let us now consider partial evaluation. Recall that a partial evaluator is a program, `mix`, which when evaluated with a given program, `pgm` and part of that program's input data ($\mathtt{data}_1$) as actual parameters yields a program which when evaluated with the rest of the programs input data ($\mathtt{data}_2$) yields the same result as the original program `pgm` would have given when applied to all its input data. A partial evaluator, `mix`, can thus be characterized by the following so-called mix-equations (note that there are the same encoding requirements for as for the self-interpreter above as the partial evaluator in essence is a smart self-interpreter). If

$$\overline{\mathtt{prg}'}^{\mathrm{Pgm}} = [\![\mathtt{mix}]\!]\ \overline{\mathtt{prg}}^{\mathrm{Pgm}}\ \overline{\mathtt{data}_1}^{\mathrm{Val}}$$

then

$$[\![\mathtt{prg}]\!]\ \mathtt{data}_1\ \mathtt{data}_2 = [\![\mathtt{prg}']\!]\ \mathtt{data}_2$$

with the same termination properties of the two sides. Note that here we use the assumption that we can remove the representation. The traditional approach to partial evaluation is illustrated in Figure 2 where the situation is generalized beyond the "two parameters, one static and one dynamic" simplification. This makes it necessary to use a binding-time description of the parameters. Please note that the illustration does not tell the full story: several phases of an actual implementation (e.g., the binding-time analysis) have been left out.

It is easily seen that a partial evaluator exists for most languages: simply let `prg'` be a copy of `prg` modified to only one parameter and to start out by binding the previous first formal parameter to its actual value $\mathtt{data}_1$. (Only trivial languages lacking, say, the ability to control the number of parameters will fail here.) It is also clear that no gain in efficiency can be expected using this method, though. In mathematics this is known as Kleene's $S_{m,n}$ theorem.

Using a self-applicable partial evaluator it is possible to obtain a compiler and a compiler generator. This is expressed by the equations 1.1–1.3 shown below, now known

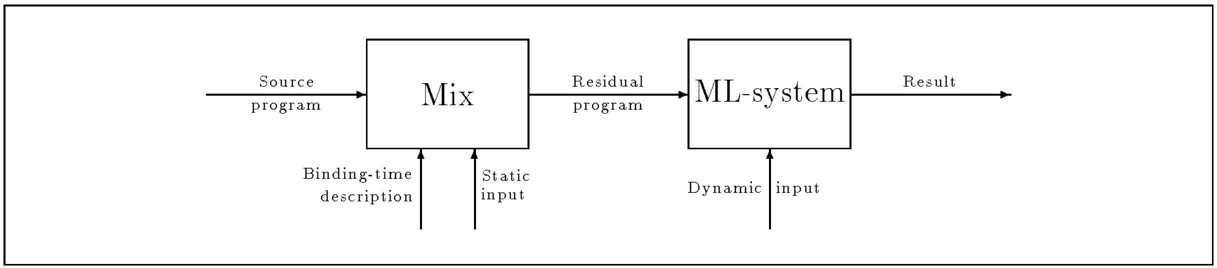

Figure 2: Two-phase evaluation by use of a partial evaluator.

as the Futamura projections. Let `int` be an interpreter for some programming language L and let `prg` be a program written in L. We then have the following provided the partial evaluator `mix` terminates.

$$\overline{\texttt{target}}^{\mathrm{Pgm}} = [\![\texttt{mix}]\!] \left(\overline{\texttt{int}}^{\mathrm{Pgm}}, \overline{\texttt{pgm}}^{\mathrm{Val}}\right) \tag{1.1}$$

$$\overline{\texttt{compiler}}^{\mathrm{Pgm}} = [\![\texttt{mix}]\!] \left(\overline{\texttt{mix}}^{\mathrm{Pgm}}, \overline{\overline{\texttt{int}}^{\mathrm{Pgm}}}^{\mathrm{Val}}\right) \tag{1.2}$$

$$\overline{\texttt{cogen}}^{\mathrm{Pgm}} = [\![\texttt{mix}]\!] \left(\overline{\texttt{mix}}^{\mathrm{Pgm}}, \overline{\overline{\texttt{mix}}^{\mathrm{Pgm}}}^{\mathrm{Val}}\right) \tag{1.3}$$

The first projection states that compilation can be done by specialization of an interpreter with respect to a program. The next equation states that a stand-alone compiler can be generated by specializing the partial evaluator itself with respect to an interpreter, and finally, the last projection expresses that a stand-alone compiler generator can be generated by specializing the specializer with respect to the specializer. Given an interpreter, `cogen` produces a compiler. The Futamura projections are more thoroughly studied, albeit in an untyped setting, in [Jon88].

The `cogen` we obtained by self-application can be used not only on interpreters but also on other programs, so we have now seen how a program generator generator can be obtained by self-applying a partial evaluator. As already mentioned, there are some problems with this approach, however. We now explain the main problem which can be observed from the Futamura equations: the double encoding of value arguments when `mix` is self-applied.

1.2.1 The Coding Problem

As mentioned above a traditional partial evaluator needs to manipulate values in order to perform some calculations. Because the number of types in `mix` itself is bounded (true for any program) and the number of possible types in input programs is unbounded, values in general have to be coded into some universal type, which could for example be defined this way

```
datatype value =
  Int of int                       (* Base type *)
| ...                              (* More bases types *)
| Record of (string * value) list  (* ML records *)
| Cons0 of string                  (* Constructor without argument *)
| Cons of string * value           (* Constructor with argument *)
| ...                              (* whatever *)
```

Base types are represented by simply adding a tag, records as lists of labels and values, and constructed values as the name of the constructor plus its argument if present. Programs are represented as their abstract syntax trees using some data type. For example we would represent the list `[10,20]` (which is a syntactically sugared way of writing the Bare Language expression `(op::){1=10,2=(op::){1=20,2=nil}}`) by

```
Cons("::",Record[("1",Int 10),
                 ("2",Cons("::",Record[("1",Int 20),
                                       ("2",Cons0 "nil")]))])
```

The doubly-coded version of the list is so large that we will not show it here, and it requires little imagination to see that a doubly-coded program (the list might be a tiny part of a program) is a gigantic constructed value. Experience shows that naïve double-encoding is prohibitively expensive for self-application — the costs are simply too high.

There are several ways out of the double-coding problem but they are best classified as "hacks." In [And92, Section 2.2.2] and [Lau91] the trick is to enhance the universal type with a subtype, program, at the same level as the ground types instead of having programs represented as a constructed value. In [dN93] the trick is to treat all values (and thus programs as well) as having one and the same "black-box" type inherited from the implementation language of the partial evaluator. This language must of course be untyped and is Common Lisp for the Petrarca case. A side effect of using this approach is that all manipulations with values must also be done in the host language, so the host language should probably not be "weaker" than the language being partially evaluated if complications are to be avoided.

The use of the described encoding gives rise to another problem in traditional partial evaluation: some unnecessary encoding and decoding operations may be left in the residual program. As an example [And92, Section 6.1] consider specialization of a self-interpreter (recall that any traditional partial evaluator contains such an self-interpreter) with respect to the power function which can be defined as follows in SML.

```
fun pow (n:int) (x:int) =
  if n=0 then
    1
  else
    (x * pow (n-1) x)
```

We might end up with something like this

```
fun pow (n:value) (x:value) =
  if val_to_int n = 0 then
    int_to_val 1
  else
    int_to_val (val_to_int x *
                (val_to_int (pow (int_to_val (val_to_int n - 1)) x)))
```

where `int_to_val` injects an integer into the universal type (so if we used the above data type it would simply be the `Int` tag) and `val_to_int` is its partial inverse (which would be `(fn (Int i) => i)`). We would have liked the resulting power program to be essentially the same as the original as it would have been for an optimal `mix`, see [JGS93, Section 6.4]. To overcome this problem, an untagging analysis is created in [And92].

1.3 Partial Evaluation Using the Cogen Approach

In this section we describe how the above described problems disappear when a program generator generator is written directly. In fact we shall see that there other benefits compared to the traditional partial evaluation approach.

Assume we have written a program generator generator `cogen` directly. To compare with the traditional approach lets consider how specialization of a program `prg` with respect to the static part of input ($\mathtt{data}_1$) proceeds. The equations are as follows (where $\mathtt{data}_2$ is the dynamic input).

$$\begin{aligned} \overline{\mathtt{p\text{-}gen}}^{\mathrm{Pgm}} &= [\![\mathtt{cogen}]\!]\ \overline{\mathtt{prg}}^{\mathrm{Pgm}} && (1.4)\\ \overline{\mathtt{p\text{-}res}}^{\mathrm{Pgm}} &= [\![\mathtt{p\text{-}gen}]\!]\ \mathtt{data}_1 && (1.5)\\ \mathtt{result} &= [\![\mathtt{p\text{-}res}]\!]\ \mathtt{data}_2 && (1.6) \end{aligned}$$

and the correctness criterion is again that

$$\mathtt{result} = [\![\mathtt{prg}]\!]\ \mathtt{data}_1\ \mathtt{data}_2$$

As is apparent from the equations, a hand-written compiler generator does not have to manipulate values of the object program — all it does is to export each part of the syntax tree to either the compiler or the target program[1] The coding problem therefore disappears in the cogen-setting and as self-application is irrelevant the double-coding problem evaporates. Moreover, when partially evaluating using a hand-written compiler generator the residual program need not inherit the universal data type for the values, so there is no need for an untagging analysis! The cogen approach to partial evaluation is illustrated in Figure 3 where again the situation is generalized. Please note that the illustration does not tell the full story: several phases of an actual implementation like ours (e.g., the binding-time analysis) have been left out.

As an aside we show that one in fact can obtain a traditional partial evaluator using the program generator generator, if we have a self-interpreter, `sint` (to simplify the presentation we here do not distinguish between a program text and its representation in the

[1] As the target program does not exist when the compiler generator is run we ought to have used a phrase like "arrange for the compiler to export..." to be precise.

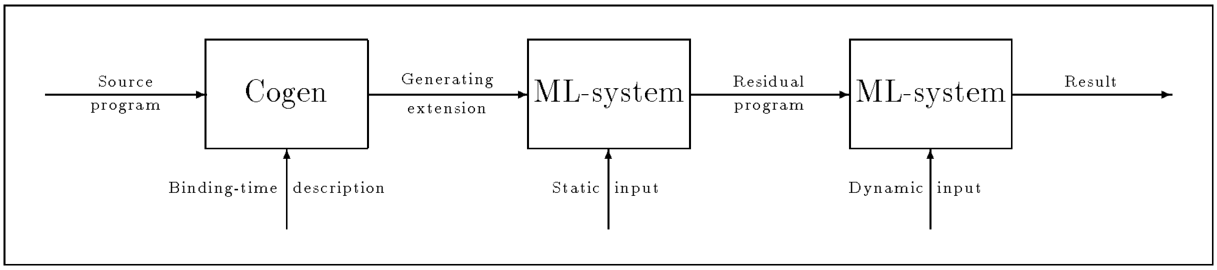

Figure 3: Three-phase evaluation via a generating extension.

data type Pgm):

$$
\begin{aligned}
\llbracket \texttt{mix}' \rrbracket &= \lambda\texttt{prg}.\lambda\texttt{data}.\llbracket\, \llbracket \texttt{cogen} \rrbracket\ \texttt{prg} \rrbracket\ \texttt{data} && (1.7)\\
&= \lambda\texttt{prg}.\llbracket \texttt{sint} \rrbracket\ (\llbracket \texttt{cogen} \rrbracket\ \texttt{prg}) && (1.8)\\
&= \llbracket \texttt{sint} \rrbracket \circ \llbracket \texttt{cogen} \rrbracket && (1.9)
\end{aligned}
$$

so syntactical composition [Jon91] of the self-interpreter and the compiler generator yields the desired partial evaluator. Here, we have used the fundamental interpreter property:

$$\llbracket \texttt{prg} \rrbracket\ \texttt{data} = \llbracket \texttt{sint} \rrbracket\ \texttt{prg}\ \texttt{data}. \qquad (1.10)$$

An example of how partial evaluation by compiler generation works in practice is appropriate: Consider again the power function defined above. Assume that the value of `n` will be known at specialization time (let us assume it turns out to be 5) while the value of `x` will not be known. The job for the compiler generator is now to produce a program, `pow-gen`, that given the value of `n` produces a program of one parameter `x` that raises `x` to the power of five.

We introduce the Lisp-like notation that the value ⌈`ml-code`⌉ is an expression that evaluates to the (syntax tree of) `ml-code` except that dot-underlined structures inside are evaluated beforehand.[2] Using this notation a compiler generator could for example generate the following output:

```
(* ... *)
fun pow-gen (n:int) (x:code) =
  if n=0 then
    ⌈1⌉
  else
    ⌈x * (pow-gen (n-1) x)⌉
(* ... *)
```

We do not know the value of `x` so we instead call `pow-gen` with ⌈`x`⌉ for `x` and of course with 5 for `n`. To obtain the result this program first recurs until `n` reaches 0. The result of that call will be ⌈`1`⌉. Using the notation that ⇒ means "evaluates to" we therefore get `(pow-gen 1 x)`⇒⌈`x*1`⌉ and `(pow-gen 2 x)`⇒⌈`x*x*1`⌉. This continues and we finally get

```
(pow-gen 5 x)⇒⌈x*x*x*x*x*1⌉
```

[2]In Lisp it is customary to use backquotes and commas, but that would be confusing when mixed with ML syntax.

This is an expression, not a function declaration, but do not worry about that; we will deal with that later. The expression is clearly as good as one usually expects from partial evaluation — there are no left-over tests. The multiplication by one could of course be removed by an algebraïc transformation, but that is not the issue here. When we bind `x` in the resulting expression to (say) 2 we get the expected value 32 just as we would have obtained 32 by calling `(pow 5 2)`.

Notice how closely the two versions of the power function resemble each other; the difference here is the insertion of $\lceil\ \rceil$ and the underlining (which is notation for limiting the scope of $\lceil\ \rceil$.) The compiler generator "only" had to decide which calculations should be done in the compiler (`pow-gen`) and which should be done in the residual program (the expression above). When evaluating 2^5 in the described three-phase way the compiler generator did none of the calculations (multiplications, subtractions, and equality tests) of `pow`, they were all done by the underlying SML-system: some at specialization time by the generating extension (the subtractions and the equality tests) and some at runtime by the resulting expression (the multiplications).

Please notice that the above is by no means the full story; we have ignored problems with code duplication, termination, and lots of other things.

Conceptually, doing partial evaluation via compiler generation is more complicated than the traditional way, as we have three "times" instead of two. The following table describes the terms we use in the two cases:

When running...	Which is...	It is called...
`mix`	hand-written	specialization time.
`res`	output from `mix`	residual or runtime.
`cogen`	hand-written	cogen time.
`comp`	output from `cogen`	specialization or compile time.
`res`	output from `comp`	residual or runtime.

The term "compile time" is primarily used when the input to `cogen` is an interpreter in the traditional sense and not when it is a program like the power function. The overlap in terms should not be a problem.

1.3.1 Other Benefits with the Cogen Approach

In this section we describe some other benefits with the program generator generator approach to partial evaluation compared to traditional partial evaluation.

Self-Application

As can be seen from the Futamura projections above, it is through self-application of the partial evaluator that a program generator generator can be obtained. Self-application of the partial evaluator puts severe constraints on the implementation of `mix`:

- The program `mix` must be written in the same language as it treats, or possibly a subset thereof. If the partial evaluator does not handle some feature of the language, say assignments, then it cannot be written using those features.

- The program `mix` must be written in a clean way with separation of binding times in mind. Whatever this means depends on `mix` itself, but it implies that constructs that are handled poorly cannot be used in `mix` itself — otherwise the partial evaluation degenerates. The point is that it is a tricky and tedious job to write `mix` in such a way that good self-application results can be obtained.

- The size of `mix` must be kept as small as at all possible, because generation of a compiler generator from a partial evaluator requires double self-application and every level of self-application expands the size of the residual program by some factor as explained above. An expansion factor of ten has been reported by [dN93].

None of these restrictions apply to a hand-written `cogen` as self-application is of no interest. Notice that the compiler generator can itself be written in another language than the language it treats. Furthermore a compiler generator can freely use "dirty tricks" like assignments, exceptions, and file input/output in the generating extension if desired. That is not possible for `mix`, unless it handles those features itself, since the compiler is a specialized version of `mix`.

Embedded Interpreter

A traditional partial evaluator performs some of the calculations in the source program right away while others are deferred until the residual program is run. When a compiler generator is used some of the calculations will be done by the compiler (the output from the compiler generator) and the rest will be done by the compiled program (the output from the compiler). This was exemplified by the power function in the previous section.

The main point here is that a traditional partial evaluator needs some kind of embedded interpreter in order to perform those calculations at partial evaluation time. It turns out that this interpreter must be rather smart about evaluating patterns or else the goal of self-application cannot be reached [Jør92, Chapter 13]. In contrast cogen does *only* code generation and no other computations at all.

Writing a self-interpreter for a language like Standard ML is a major project, and if we had had to do it we would have been able to put less work into other parts of our thesis.

Chapter 2

Standard ML

language (lang′gwij), *n.*
[...] **3.** *the system of linguistic signs or symbols concidered in the abstract (opposed to* speech*).* **4.** *any set or system of such symbols as used in a more or less uniform fashion by a number of people, who are thus enabled to communicate intelligibly with one another.* **5.** *any system of formalized symbols, signs, gestures, or the like, used or conceived as a means of communicating though, emotion, etc.* [...]

— WEBSTER'S UNABRIDGED, *1989 edition (1989)*

The programming language Standard ML is a stratified language in the sense that it can be divided into a Core Language and a Modules Language. ML has been formally defined in the Definition of Standard ML [MTH90] and the definition has been explained and properties of the language have been proved in the Commentary on Standard ML [MT91].

About half of the phrase classes of the Standard ML Core Language are derived forms, whose meaning is defined by translation into the other half which is called the Bare Language ([MTH90, Appendix A] defines the translation of derived forms.) Sections 4 and 6 of the Definition of SML define the static and dynamic semantics of the Bare Language by natural semantics.

We will only be concerned with the Core Language in this thesis. As in [MTH90] this can and will be done by considering the Bare Language only (in examples we will, however, allow ourselves to use Core Language syntax also for notational convenience.) We assume the reader is familiar with the syntax and the static and dynamic semantics of the Core and Bare Language and we will use the terminology of the Definition of Standard ML, but review some of the more unfamiliar aspects when needed. We are mostly interested in the dynamic semantics of SML in this work, and just as the dynamic semantics of SML is only defined for declarations that satisfy the static semantics, we will only consider partial evaluation of well-typed programs.

At this time our methods cannot be used to handle the full Standard ML Core Language, so we have made three restrictions on the programs to be treated. These three restrictions will be explained and motivated in Section 2.1.

To avoid unnecessary work we show in Section 2.2 how Core SML programs obeying the three restrictions can be simplified by preprocessing without restricting the class of Core SML programs we can handle.

2.1 Restrictions on SML Input Programs

The definition of the language states that SML is an interactive language in the sense that function declarations and expressions can be evaluated by the user at any time in an interactive session. We do not treat this aspect of SML as off-line approaches to partial evaluation need to be able to analyze the whole program. Instead we view a Core Language program as one declaration defining essentially one input-output function (called the main function), and we expect the user of the partial evaluator to provide the name of this function and a binding-time description of the arguments to this function.

We will not consider the nondeterministic `Interrupt` exception. This exception is raised when the user requests that evaluation be stopped. The reason for this restriction is simple: no program manipulations can be semantics preserving when `Interrupt` can be raised and handled.

Our methods are not strong enough to handle polymorphism, so we will require that the main function is monomorphic or alternatively that the user provides a monomorphic type for it. Having made this restriction, preprocessing can duplicate code so that all functions are monomorphic after preprocessing. Let us take a brief look at the problems with polymorphism in partial evaluation by considering the following function

```
fun equal x y =
  (x=y)
```

Let us assume that the value of `x` is known while the value of `y` is not. Let us furthermore assume that `equal` is the main function we want to specialize. The SML-type of `x` can among other things be both *int* and *bool*. The problem now is that the value of `x` must be put into the residual program (i.e., it must be "lifted" using a term from [GJ89]) and to do that we have to put some piece of code into the generating extension of `equal` that does the lifting. To do this in a well-typed manner[1] we must give the generating extension for `equal` an extra parameter that somehow describes the type of `x` (one cannot make the lift function polymorphic since it has to generate different pieces of code for different types of arguments). That extra parameter might be a function or a tag injecting the relevant type into some universal value type.

```
fun equal-gen x type_of_x y =
  ⌈lift (type_of_x x)=y⌉
```

While this approach solves the problem with polymorphism in input programs it also leads to less efficient generating extensions and one should make an analysis to eliminate

[1] In languages like Haskell it would be possible to use overloaded functions. The use of overloading would, however, just be a cosmetic solution as the check needed at specialization-time still makes the generating extension slower.

as many as possible of the extra parameters introduced. Note that this situation closely resembles the tagging problem as described in for instance [And92, Section 6.1]. Note also that the problem here is not restricted to the special equality predicate — polymorphic constructors like `::` ("cons") would have created the same problems.

As we shall see in Section 3.2 we want to keep track of which values a function has been specialized with respect to. This would also require some kind of tagging and we have explicitly tried to avoid coding values, as explained in Section 1.2.1.

Having explained why we have chosen to avoid polymorphism we now turn to the consequences. The request for all function to be monomorphic is less restrictive than it sounds since intraprogram polymorphism can be eliminated by unfolding, or it can be classified as dynamic (in which case it does not pose any problem); this way, the only real restriction is that the top-level function to be specialized must be monomorphic. The question now is: is unfolding practically useful or do the unfolded programs become excessively large? The answer to this of course depends on programming style, but more can be said. The worst case is exponential increase in size, but we conjecture that the normal case is linear. Support for this can be found in [Jon93] where the size increase is reported to be 43%–133% for three large programs.

* * *

Apart the from the restrictions just described the techniques in this thesis can be applied to any Core SML program. Exceptions, assignments, and input/output are all handled although the latter two are just postponed to the residual program.

2.2 The Simplified Bare Language

We will now describe how a Bare Language program obeying the restrictions described in the preceding section can be transformed, without changing its semantics, by simple preprocessing into what we call a Simplified Bare Language program. The syntax for this Simplified Bare Language is like the syntax of the Bare language, just with some further syntactic restrictions. That is, we shall use the same grammar as the one presented in Figures 3 and 4 in the Definition of SML [MTH90], but add some syntactic restrictions to the list of syntactic restrictions in [MTH90, Section 2.9]. The reason for this approach, as opposed to defining a new grammar expressing the syntactic restrictions by grammar rules, is that the rules for static and dynamic semantics in Sections 4 and 6 of the Definition of SML then also apply without modification to our Simplified Bare Language.

To begin with we note that it is possible to alpha-convert a Bare Language program in such a way that different identifiers have different names. We will refer to this process as alpha-conversion and we will expect it to make sure that user defined names do not coincide with the names we generate in the generating extension. There are lots of reasons to alpha-convert programs: it eliminates the need for an explicit environment in the binding-time analysis, it reduces problems with variable capture, it makes local declarations equivalent to sequential declarations, and it simplifies the structure of the generating extension.

All matches can be made exhaustive and redundant rules can be removed. If a match occuring in a lambda expression is non-exhaustive, it is made exhaustive by adding a match rule `_ => raise Match`. A match in a handle expression is made exhaustive by adding the match rule `x => raise x` instead.

We will now go into details of the grammar and the reader should follow the grammar of Figures 3 and 4 of the Definition of SML. We will cut away productions that we do not need from that grammar. It should be noted that the majority of the simplifications will essentially not interfere with the way a program is evaluated.

`open`-declarations and long identifiers are not needed as they are for dealing with modules. Infix/nonfix declarations and uses of `op` are in a sense syntactic sugar which the parser should eliminate. Type ascriptions as for example in *exp* : *ty* are not needed due to the monomorphism present at this stage; type declarations (*not* data type declarations!) can be unfolded.

All data type declarations, including those locally declared, can be moved to the beginning of the declaration of which the program consists.

An abstype declaration `abstype` *datbind* `with` *dec* `end` can be transformed into a sequential declaration `datatype` *datbind dec*, and the data type declaration can be moved as any other data type declaration as described above. This is due to well-typedness and alpha-conversion.

Exception declarations are also generative in Standard ML but can with some trouble also be moved out of local declarations. How to do this and why it causes trouble is described in Section 3.9. Further simplifications are also discussed there.

Local declarations can be replaced by sequential declarations, as alpha-conversion has already made sure that no identifier conflicts will arise and because data type and exceptions (the generative constructs) in the local scope have been eliminated.

Non-recursive single value bindings have the form *pat* = *exp*. They can be transformed into a sequential declaration where the pattern on the left hand side of the value binding consists of a variable only. The motivation for this simplification is simply that we want to concentrate all pattern matching in the lambda expressions to keep the analysis simple. For example, the value binding `val C(1,x,y) = ...` can be transformed into

```
val temp = ...
val x = case temp of
            C(1,x,y) => x
          | _ => raise Bind
and y = case temp of
            C(1,x,y) => y
          | _ => raise Bind
```

This, of course, introduces some inefficiency as the pattern matching is performed twice instead of once. We believe that the extra cost is small enough for this not to be a practical problem; experience will show whether this is true. Note that the transformation implies that exception `Bind` can only be raised explicitly by an expression and not by a declaration in itself.

Wildcards in pattern rows (three dots) can be removed due to well-typedness, since for any well-typed SML program the set of labels corresponding to any occurrence of ... in a pattern row is uniquely determined. The precise rules for well-typedness of flexible records in SML are given in the Definition and in the Commentary. If the set of labels corresponding to an occurrence of the pattern row wildcard is $\{lab_1, \ldots, lab_n\}$ then the wildcard can be replaced by the pattern row $lab_1 = \_ , \ldots ,\ lab_n = \_$ (where the ellipsis is notation for the middle $n-2$ patterns — not another pattern row wildcard.)

Also, any expression row, pattern row and type-expression row can be permuted so the labels are sorted lexicographically. This is semantics preserving and the actual labels become irrelevant.

Wildcard atomic patterns (underscores) can be transformed into variable atomic patterns by using fresh variables. Note that this should not have any bad effects on efficiency since any reasonable compiler should be able to do the "inverse operation," i.e., to insert wildcards for pattern variables that are not used in a corresponding expression.

* * *

atexp	::=	*scon*	special constant
		var	value variable
		con	value constructor
		excon	exception constructor
		`{` ⟨*exprow*⟩ `}`	record
		`let` *dec* `in` *exp* `end`	local declaration
		`(` *exp* `)`	
exprow	::=	*lab* `=` *exp* ⟨ `,` *exprow*⟩	expression row
exp	::=	*atexp*	atomic
		exp atexp	application
		exp `handle` *match*	handle exception
		`raise` *exp*	raise exception
		`fn` *match*	function
match	::=	*mrule* ⟨ `|` *match*⟩	match
mrule	::=	*pat* `=>` *exp*	match rule
dec	::=	`val` *valbind*	value declaration
		`datatype` *datbind*	datatype declaration
		`exception` *exbind*	exception declaration
			empty declaration
		dec_1 ⟨;⟩ dec_2	sequential declaration

Figure 4: Syntax of expressions, matches, and declarations (programs).

Figures 4 and 5 show the grammar we use; the start symbol is *dec*. The reader should familiarize himself/herself with the names of the productions as they will be used extensively in the following chapters. We remind the reader that the angular parentheses, ⟨like these⟩, are used for optional components and that special constants are integers, reals, and strings.

valbind	::=	*pat* `=` *exp* ⟨`and` *valbind*⟩	plain value binding
		`rec` *valbind*	recursive value binding
datbind	::=	*tycon* `=` *conbind* ⟨`and` *datbind*⟩	data type binding
conbind	::=	*con* ⟨`of` *ty*⟩ ⟨ `|` *conbind*⟩	constructor binding
exbind	::=	*excon* ⟨`of` *ty*⟩ ⟨`and` *exbind*⟩	exception binding
atpat	::=	*scon*	special constant
		var	variable
		con	constructor
		excon	exception constructor
		`{` ⟨*patrow*⟩ `}`	record
		`(` *pat* `)`	
patrow	::=	*lab* `=` *pat* ⟨ `,` *patrow*⟩	pattern row
pat	::=	*atpat*	atomic
		con atpat	value construction
		excon atpat	exception construction
		var `as` *pat*	layered
ty	::=	`{` ⟨*tyrow*⟩ `}`	record type expression
		tycon	type construction
		ty `->` *ty′*	function type expression
		`(` *ty* `)`	
tyrow	::=	*lab* `:` *ty* ⟨ `,` *tyrow*⟩	type-expression row

Figure 5: Syntax of bindings, patterns, and type expressions

2.3 Notational Conventions

Programs written in the (Simplified) Bare Language look different from "normal" Standard ML programs since they cannot use the many syntactic conveniences of the Core Language. For the sake of readability we will allow ourselves to write example programs in the Core Language, so we will write

```
fun max (x,y) = if x>y then x else y
```

and not

```
val rec max = fn {1=x,2=y} =>
  (fn false => x | true => y) (op> {1=x,2=y})
```

unless of course we are making a point (as here). The example here shows the three most common derived forms: "`fun`," tuples, and "`if`." Sometimes we will even use comments in example programs.

Also, we will not be concerned with the resolving of overloading, except in Section 3.8.2 where we specifically address that problem. For instance the `max` function above is not a well-typed SML program (one cannot tell from the declaration whether `>` is to be applied to a pair of integers or a pair of reals.) Henceforth the reader can simply imagine that all overloaded arithmetic operators operate on integers as reals and integers are treated alike. We will also assume the existence of an omnipresent polymorphic data type, `Option`, defined the following way:

```
datatype 'a Option = Some of 'a | None
```

The data type is used as a wrapper for other data types, allowing a value of such a types to be present af `Some x` or absent as `None`. We will use this data type in both compiler generator and generating extensions and even in some meta-functions in this thesis.

Chapter 3

Specialization of Simplified Bare Language

"Thou mayst go thy ways, my friend," said the Captain, addressing Gurth, in special confirmation of the general voice, "and I will cause two of my comrades to guide thee by the best way to thy master's pavilion, and to guard thee from night-walkers that might have less tender consciences than ours; for there is many one of them upon the amble in such a night as this. [...]"

— WALTER SCOTT, *Ivanhoe (1895)*

In this chapter we study specialization of the Simplified Bare Language and how to generate a generating extension for a given program which performs the specialization we wish. The basic technique of specialization is *polyvariant program point specialization*, where a program point is equivalent to the name of a declared function.

This presents some new and challenging problems. First of all, as remarked by Jones, Gomard and Sestoft [JGS93, page 11] it is "far from clear how to construct [a program generator generator]." Second, due to the complicated nature of the language we treat, i.e., the presence of user-defined possibly recursive data types, exceptions, side effects and complicated pattern matching, program specialization is more complicated than in earlier partial evaluators for mostly functional languages.

The goal of this chapter is to analyze the new problems; we believe, however, that many of the observations can be used in the development of compiler generators for other languages. The next chapter presents a formal definition of the compiler generator for the Simplified Bare Language — defined via a *two-level* semantics.

As already noted, the cogen approach to specialization is an *off-line* strategy, where the generation of the generating extension is done on the basis of an a priori performed binding-time separation. In Chapter 5 an automatic binding-time analysis for the Simplified Bare Language is developed; for now we will assume that all phrases have been annotated in some unspecified manner and that binding times for all phrases are available. In the next chapter the annotation is formally defined via a definition of a Two-Level Simplified Bare Language. Recall that we only consider specialization of well-typed programs;

we may therefore assume that type-information is available for every single phrase during generation of the generating extension.

Overview of the Chapter

As this chapter is long and because especially its second half is not characterized by linear development, we sketch an outline of the chapter here.

- We start out by defining our goals and describing our basic decisions that leave their stamps on the entire process of specialization.
- What should the generating extensions look like? We discuss two very different possibilities.
- The generating extensions produce programs so we turn to how they can represent and manipulate code.
- Functions, including recursive ones, may be declared locally. We discuss the consequences of these nested scopes and in particular what to do with `let`-expressions.
- Standard ML is permeated with pattern matching, so we discuss building, decomposition, and matching of structured values. In the course we have to consider the possibility of structures which are only partially known at specialization time.
- For completeness we then discuss how to handle miscellaneous features of the language.
- Lastly, we discuss exceptions and show how to tame them for partial evaluation.

3.1 Goals and Fundamental Design Decisions

In this section we will make some fundamental design decisions concerning the specialization. They are made here to establish a framework for the rest of this chapter. This also allows us to recall some of the fundamental concepts of specialization. We will be brief as most of the problems and considerations are known and refer the reader to the literature for more thorough discussions.

3.1.1 Goals

The guideline for our design decisions is that we wish to focus on the problems with the cogen approach and to be able to treat the full Standard ML Core Language, with only the restrictions described earlier in Section 2.1. We are willing to give up specialization power to meet these goals. Another goal is semantic safety in the sense that not only should the residual program give the same result as the original, but the residual program should also never be less efficient than the original program, and the termination properties of the residual program compared to the original program should not be changed. (Having unchanged termination properties is the goal, but in "real life" this is relaxed a bit: the generated residual program must never terminate for values that the source program does not terminate.)

3.1.2 Polyvariant Program Point Specialization

We will make use of the basic technique of polyvariant program point specialization, which means that a program point in the source program can be specialized into several program points in the residual program [Jon88].

3.1.3 Mono- or Polyvariant Binding Times

If all phrases are assigned one and only one binding time, the assignment of binding times is called *monovariant.* If phrases are allowed to be assigned a set of binding times (where each binding time in the set corresponds to a different context) the assignment of binding times is called *polyvariant* [JGS93], not to be confused with polyvariant program point specialization. It is obvious that a polyvariant separation of binding times may give better specialization, but it complicates the specialization process. Polyvariant binding times are perhaps especially relevant for functional programming languages since functional programs often use quite a lot of general library functions which may be called in different binding time contexts, i.e., with different combinations of static and dynamic arguments, without one context affecting the others.

However, there are some problems with using polyvariant binding times: it may lead to a very large number of variants for each function; if a function has n parameters each of which can be either static, `S`, or dynamic, `D`, then we may up to get 2^n variants. This problem is not only theoretical: in [dN93, Sections 6.8 and 9.5] it is reported that the number of variants is prohibitive for successful self-application of the Petrarca specializer. Even worse, it turns out that many of the generated variants are essentially the same. Also polyvariant binding-time analysis is more complicated than monovariant binding-time analysis; while the monovariant algorithm we will present is almost linear in time-complexity, polyvariant algorithms so far have been cubic-time algorithms or worse.

For these reasons we will use monovariant binding-time assignments.[1]

3.1.4 Specialization with Respect to Equality-Types

During specialization of a function with respect to some value, we want to be able to check whether or not the function has been specialized with respect to this value before (to avoid code-duplication). This must somehow involve using the equality predicate of Standard ML, which is only applicable to equality-typed values [MTH90, Section 4.5]. In other words, if we want to specialize with respect to higher-order values we need to represent the values and check for equality of representations rather than of the values themselves.

[1] Except for the pervasive functions: it should, of course be possible to perform additions during specialization even though one addition operation is deferred to residual time.

This is the approach taken in Similix [Bon91], and as noted by Launchbury [Lau91] it shows that even in the untyped world one cannot in general avoid encoding values.

Because one of the motivations for this work was to avoid manipulating representations of values, and because specialization with respect to higher-order values is complicated (we shall discuss some of the problems in Section 3.4), we choose to specialize with respect to equality-typed values only.

3.1.5 Unfolding

To improve the quality of the residual program by eliminating expensive and unnecessary function calls we must arrange for as many calls as possible to be unfolded. Unfolding can be performed during specialization or as postprocessing or even both. Since not performing any call unfolding during specialization at all tends to produce unreasonably many residual functions we choose to perform call unfolding on the fly. This unfolding has the additional benefit that it lowers the cost of not specializing with respect to higher-order values. This is because only higher-order parameters to calls that are not unfolding need to be made dynamic.

An unfolding strategy must, as described in [JGS93, Section 5.5], avoid infinite unfolding and avoid duplication of code and computation. Moreover, computations which involve side effects or exceptions should not be reordered or changed in number, as reordering of such expressions in general, of course, is not semantics preserving. Also, side effects should neither be discarded nor duplicated.

Avoiding Duplication, Reordering and Discarding

In a strict language like Standard ML one can separate the problem of infinite unfolding from those of duplication, reordering ,and discarding. This is done in Similix [BD91] [JGS93, Chapter 10] by inserting `let`-expressions prior to specialization. We will adopt that strategy. Hence `let`-expressions are to be inserted for `fn`-expressions in the following way. Recall that a `fn`-expression has the form

$$\begin{array}{llll} \texttt{fn} & pat_1 & \texttt{=>} & exp_1 \\ \;\texttt{|} & \ldots & & \\ \;\texttt{|} & pat_n & \texttt{=>} & exp_n \end{array}$$

where $n \geq 1$. `let`-expressions are then inserted around each exp_i, where each `let`-expression rebinds the variables bound in the corresponding pattern pat_i to themselves. That is, if we let $x_{i1}, \ldots, x_{ij(i)}$ denote the set of variables in the pattern pat_i, then the above `fn`-expression is transformed into

$$\begin{array}{llll} \texttt{fn} & pat_1 & \texttt{=>} & \texttt{let val } x_{11} \texttt{ = } x_{11} \texttt{ and } \ldots \texttt{ and } x_{1j(1)} \texttt{ = } x_{1j(1)} \texttt{ in } exp_1 \texttt{ end} \\ \;\texttt{|} & \ldots & & \\ \;\texttt{|} & pat_n & \texttt{=>} & \texttt{let val } x_{n1} \texttt{ = } x_{n1} \texttt{ and } \ldots \texttt{and } x_{nj(n)} \texttt{ = } x_{nj(n)} \texttt{ in } exp_n \texttt{ end} \end{array}$$

The insertion of these `let`-bindings means that a function call can be unfolded without the risk of duplication, reordering, or discarding. The principle is that it is safe to duplicate a variable so we introduce *protected variables* for the expression. We refer the reader to [JGS93] for details.

In Similix an *abstract occurrence count* analysis is employed to detect when the `let`-expressions can be unfolded. A `let`-expression is allowed to be unfolded if and only if the variable bound (in the subset of Scheme treated by Similix, each `let`-expression only binds one variable) will occur exactly once in the residual program. Note that if unfolding were allowed whenever a variable would not occur in the residual program, we could risk discarding computations and so change program semantics.

We expect Standard ML compilers to be smart enough to unfold `let`-expressions (safely) without our assistance. We will therefore not attempt to unfold any dynamic `let`-expressions. A different approach is used in Similix for two reasons: 1) the residual programs become more readable when excess `let`-expressions are unfolded, and 2) Scheme is often interpreted in a naïve way. Only the first reason is applicable in our case, but we have chosen to concentrate on more interesting problems.

Avoiding Infinite Unfolding

In Similix a *specialization point* is inserted for every dynamic conditional and every dynamic lambda expression [BD91]. The specialization point is a new function, in [JGS93] termed an *sp-function.* The parameters of an sp-function are the free variables of the conditional or the lambda expression, and the body of the sp-function is the conditional or the lambda expression. The difference between an sp-function and a normal function is that a call to an sp-function is not unfolded but residualized as a call to a specialized function. Compared to traditional partial evaluators as discussed in [JSS89] and [Lau89] the central concept using this strategy is not function call annotation, but function declaration annotation.

We adopt Similix's unfolding strategy with changes caused by differences in language. In the Simplified Bare Language a conditional corresponds to (an application of) a lambda-expression with several match rules: an `if`-expression is a derived form which is transformed into an application of a `fn`-expression with two match rules. Dynamic conditionals are then `fn`-expressions with at least two rules, but without enough static information available to guarantee that the right branch can be chosen at specialization time. We shall later see that it is not trivial to find out whether one can determine which branch to choose during specialization.

We argue for inserting specialization points for dynamic `fn`-expressions and for `fn`-expressions with two or more match rules where one cannot determine which match rule to choose during specialization as follows. For exactly these two kinds of phrases the standard semantics is less strict than the specialization semantics (due to the simplifications in 3.9 matches for `handle`-expressions have exactly one match rule.) For instance, during standard evaluation only one of the match rules is chosen whereas specialization will be performed for all the match rules. Hence by employing recursion one easily writes down expressions for which standard evaluation terminates but specialization fails to terminate due to infinite unfolding unless one stops the unfolding somehow, and this is exactly the role of the specialization functions. To emphasize: all calls to user-defined functions are unfolded whereas calls to sp-functions are never unfolded.

The reason for inserting specialization points for dynamic `fn`-expressions and for dynamic conditionals is that the only non-strict contexts in the Simplified Bare Language are the right-hand sides of matches. The simplifications in 3.9 specify that matches for `handle`-expressions have exactly one match only consisting of a variable. That leaves us with `fn`-expressions as the only non-strict contexts under dynamic control. It should be noted that termination is not guaranteed by this approach as there are problems with static loops and infinite specialization; we will return to this in section 3.1.6.

We cannot allow ourselves blindly to parameterize every free variable at a specialization point because we do not specialize with respect to higher-order values. If we abstract over such a variable it would have to be dynamic with severe consequences to the residual program. Consider this example:

```
(* Apply f to all elements of xs that satisfy p *)
(* point 1 *)
fun apply f p xs =
  let
    (* point 2 *)
    fun filter [] = []
      | filter (x :: xs) =
          if p x then
            x :: (filter xs)
          else
            (filter xs)
    fun map [] = []
      | map (x :: xs) = (f x) :: (map xs)
  in
    map (filter xs)
  end
```

Assume here that `f` and `p` are known functions while the list `xs` is a list of known length but with unknown elements. That makes the `if` dynamic and we insert an sp-function. All other matches are static. If we insert the sp-function at point 1 we must pass `p`, `x`, `xs`, and `filter` as parameters, but if we pass `filter` as a parameter then it must be dynamic as we do not specialize with respect to higher order values — specialization degenerates. If, however, we insert the sp-function at point 2 (using an `and` clause since `filter` is recursive) we only have to pass `x` and `xs` as parameters since the other variables also are present in the new scope and need not be specialized for. After insertion of sp-functions the above program becomes:

```
(* Apply f to all elements of xs that satisfy p *)
fun apply f p xs =
  let
    fun sp_1 x xs =
      if p x then
        x :: (filter xs)
      else
        (filter xs)
    and filter [] = []
      | filter (x :: xs) = sp_1 x xs

    fun map [] = []
      | map (x :: xs) = (f x) :: (map xs)
  in
    map (filter xs)
  end
```

Consequently we insert an sp-function as close as possible to its origin. If the origin is within a recursive value binding, we insert the sp-function in the same value binding using an `and`-clause so they can call each other. If the origin is within a non-recursive value binding, we must instead insert the sp-function in a new value binding immediately preceding the origin's. That way it can be called and it is no problem that the sp-function cannot call the function it originates from.

The variables we abstract over are the free variables at the specialization point, except those of function type that are still in scope where the sp-function is inserted.

3.1.6 Termination

In the preceding section we have already briefly touched upon the problems of termination; however, not all kinds of termination problems are solved by our choice of unfolding strategy.

Ideally the compiler generator should terminate for any possible input program and so should the generated compiler and then its output should terminate exactly as often as the original program did. There are four different sources of non-termination, of which we will rule out one:

- The construction of the generating extension might never terminate.
- Specialization might fail to terminate because it is more strict than normal evaluation.
- Specialization might fail to terminate because the residual program needed turns out to be infinite.
- The source program might contain an infinite loop that is always reached (given the actual static values).

There are no problems with ensuring termination of the compiler generator for any possible input program, because, as it will be clear from Chapter 4, the compiler generation process in essence consists of a single pass over the abstract syntax tree for the source program.

The unfolding strategy we have chosen ensures that the specialization process only fails to terminate if the generating extension enters an infinite static loop ([JGS93, Page 118], but in this case standard evaluation also fails to terminate (if the expression is ever reached) so in this case we accept non-termination of the generating extension. Determining whether a certain program point is ever reached is halting-problem equivalent and thus undecidable. Experience shows that safe approximations make partial evaluation degenerate.

A more serious problem is the problem of infinite residual programs. Consider the following tail-recursive version of the length function for lists, where the parameter is assumed dynamic.

```
fun length l = length' l 0
and length' [ ] n = n
  | length' (_::rest) n = length' rest (n+1)
```

If we use standard binding-time analysis on the example we get the result that the accumulating parameter is static and since it grows unboundedly we will get an infinite number of residual functions. This problem has been addressed in the working note [Hol88] and in [BJ93]. The key to almost taming this problem is to collect information about which parameters that are used to determine the flow of control. One should not specialize with respect to a variable (as `n` above) that is not used for determining control. This "poor man's generalization" solution is not a real solution in the sense that it is only a heuristic, which does not *guarantee* termination of specialization. In [JGS93, Chapter 14] a binding-time analysis is developed which solves termination problems for a simple flow-chart language. (For the interested reader, [JGS93] contains more references to the literature on termination problems.)

As it is a project in itself to solve the termination problems satisfactorily for a language like SML, we will not employ any other strategy than the unfolding strategy for ensuring termination of the specialization process.

3.2 Control Structure of the Generating Extension

In this section we study the control structure of the generating extension. From now on we will also informally use the term "generating extension" of a function (and not only of a program). The generating extension of a user-defined function `f` in a program `p` is the piece of code in the generating extension of `p` corresponding to `f`.

In this section we describe two different methods of keeping track of which functions need to be specialized to which values. The first one is due to Carsten Kehler Holst and John Launchbury whereas the second, which in our case is advantageous, is original to the best of our knowledge.

The underlying principle is that the generating extension of a function should compute a specialized piece of code. To describe the two different methods we use this version of Ackermann's function as example:

```
(* Source program *)
fun ack m n =
  if m = 0 then
    n+1
  else
    if n = 0 then
      ack (m-1) 1
    else
      ack (m-1) (ack m (n-1))
```

and we assume that the value of `m` is known and the value of `n` is not. As a consequence of the monovariant binding-time assignment the first recursive call of `ack` will not be evaluated even though both parameters will be known at specialization time.

3.2.1 Functional, Breadth-First Version

The first control structure method is a purely functional method suggested in the working note [HL92]. It could also be characterized as *breadth-first* compiling as the list of further work to do is organized as a queue. The main principle is that there is an interpreting loop very much like the one used by a standard traditional partial evaluator, see [Lau91, pages 132–133]. This loop keeps a list of the specialization points (functions specialized so far and the values they were specialized to) and a similar list of functions and values of work that still need to be done. These lists are traditionally called "the done list" and "the pending list."

Whenever the main loop decides that some function `f` needs to be specialized to some values it calls the generating extension of `f` with those values. If the specialization of `f` should entail specialization of other functions (or of `f` itself with respect to other values) called inside `f` it leaves marks in the syntax tree that it returns. Those marks are searched for by the main interpreter when it regains control.

The main loop in [HL92] consists of two mutual recursive functions and looks like this (with notation changed to SML) for the Ackermann function:

```
(* Interpreter loop of functional generating extension *)
fun spec_b pend done =
  case pend of
    [] => []
  | (p::rest) =>
      if member p done then
        spec_b rest done
      else
        (case p of
           Fun_ack m => spec2_d rest (p::done) "ack" [Num m] ["n"]
                          (reduce_ack m [Parm "n"]))

and spec2_d pend done f stats parms exp =
  (f, stats, parms, exp) :: (spec_b (pend @ (search exp)) done)
```

In this and the following example the constructors of the program are exactly those identifiers with an upper case initial letter. The "`case p of`" test is redundant here but in general a test is needed in order to know which function to specialize. The generating extension for `ack` looks like this:

```
(* Generating extension, functional version *)
fun reduce_ack m p =
  if m = 0 then
    Prim "+" [p, Val (Num 1)]
  else
    If (Prim "=" [p, Val (Num 0)])
       (Call (Fun_ack (m-1)) [Val (Num 1)])
       (Call (Fun_ack (m-1)) [Call (Fun_ack m) [Prim "-" [p, Val (Num 1)]]])
```

To elaborate: `if` is a language construct that actually tests some condition while `If` is a curried constructor[2] (taking three arguments) that will eventually be printed as an `if`-expression. In other words `if` is the static conditional and `If` is the dynamic. The constructor `Call` is for constructing residual calls. Static calls are only made to primitives as `=` in the example. A call to `reduce_ack` with arguments `m=2` and `p=(Var "n")` would evaluate to

```
If (Prim "=" [Var "n", Val (Num 1)])
   (Call (Fun_ack 1 [Val (Num 1)])
   (Call (Fun_ack 1) [Call (Fun_ack 2) [Prim "-" [Var "n", Val (Num 1)]]])
```

The main benefit of this control structure is that it is purely functional and since it does not use higher order functions it could be used for almost any language, even C and Pascal. (The claim of being functional still holds for say C, since structures are built and examined but never taken apart and changed.)

The negative side of the method is that there is what we might call a left-over part of a partial evaluator in the generating extension: the program still explicitly keeps track of both pending and done lists and even searches expressions for calls to add to the pending list. This is done by the `(search exp)` above.

We remark that the working note where this method came from used *duovariant* binding times (meaning "having two variants": one for all static parameters and one for the general situation) and the first recursive call would have been completely evaluated. That is not a characteristic feature of the method, so we have made it monovariant.

3.2.2 Imperative, Depth-First Method

The second method does not have any main control loop and has no explicit pending list; instead the call stack is used. This method is characterized by its *depth-first* specialization.

Every generating extension keeps track of the static values with respect to which it has specialized so far. It does so by having an associated list variable, called "`SeenB4List`," that can be destructively updated. For efficiency reasons it is best to have one list for each function rather than a global list; it saves the use of a tag such as `Fun_ack` above (which is necessary for type-reasons since lists are homogeneous in SML), and makes the lookup operations in the `SeenB4List` faster.

Calling the generating extension with some static parameters and code for the dynamic ones "tells" the extension to specialize the original function to the given static values and to return an expression that calls the residual function with the dynamic values.

[2] I.e., a constructor as in LML or Miranda.

The generating extension first checks whether the specialization has already been done. If it has, the old residual function is used and if not, the generating extension puts the new values into the `SeenB4List` and does the specialization, which may lead to calling other generating extensions or even itself. That it may (directly or indirectly) call itself is the reason why `SeenB4List` must be updated first.

The generated residual function definitions are not returned; instead they are added to some accumulator variable. To avoid scope problems we have chosen to use one code accumulator variable per generating extension (we elaborate on this issue in Section 3.4).

With this approach the generating extension of the Ackermann function looks like this:

```
(* Generating extension, imperative version *)
val SeenB4List = ref ([]: ...)
val code = ...
fun ack_gen m n =
  let
    val (seen,name) = seenB4 SeenB4List m
    val _ = (* discard result -- side effects! *)
      if seen then
        ()
      else
        emit(code,name,["n"],
          if m = 0 then
            ⌈n + 1⌉
          else
            ⌈if n = 0 then
              ack_gen (m-1) ⌈1⌉
            else
              ack_gen (m-1) (ack_gen m ⌈(n-1)⌉)⌉)
  in
    ⌈name n⌉ (* Application not underlined *)
  end
```

Note the use of recursive backquoting; the ⌈`(n-1)`⌉ is used as a parameter for the innermost recursive call. `seenB4`, which is not shown here, takes care of both searching in and updating of the `SeenB4List`. The function `emit` is used to add a residual function, named `name`, to the code accumulator `code` with formal parameter `n` and body equal to the fourth argument of `emit`.

This method is superior compared to the method presented in the preceding section when the language, like SML, allows destructive updating of variables. The method is not particular well-suited for functional implementation. It would require passing a code list around for every generating extension function and as the `SeenB4List` would also have to be passed around as argument, it would furthermore reinvent the need for a tag for the static values. However, this method would still be preferable compared to the breadth-first method as there is no need for the interpreter loop and especially no need for searching every residual expression for calls to add to the pending list.

Standard ML evaluates depth-first like C and Scheme do and so, unlike some functional languages, evaluation order has a real significance. If the partial evaluator tracks assignments (which our system does not) or exceptions, specialization of Standard ML must therefore be depth-first so the evaluation order is not changed. Tracking assignments in a breadth-first partial evaluator would simply be a nightmare, and dealing with exceptions in a clean way would require some kind of backtracking since we might unnecessarily have specialized any number of functions to any number of values before evaluating a `raise` phrase. Consider the example

```
fun main = (deep 0, nasty 1)
and nasty x = ... (* code causing a lot of specialization *)
and deep x = deeper x
and deeper x = 42 div x
```

A breadth-first specializer would start specializing `nasty` which a depth-first specializer would never even consider because the division by zero (which raises exception `Div`) prevents the call from being made.

Hence we choose to use our depth-first control structure method presented in this section for the generating extensions.

3.3 Representation of Code in the Generating Extension

The generating extension should produce a residual program when evaluated, hence we must be able to represent SML programs in the generating extension. We choose to use an SML data type that corresponds directly to the grammar of the Bare Language to represent code in the generating extension. For the sake of concreteness in examples we present this data type in Figures 6 through 8. The data type presented is almost identical to the one used in the ML Kit [BRTT93] and also used in our implementation. It is worth noting that the language is larger (at least grammar-wise) than the language we treat (Figures 4 and 5).

3.4 Scope — Let-Expressions

In this section we discuss some scope problems that can occur when specializing a block structured higher-order language like SML and the Simplified Bare Language where it is possible to declare recursive functions locally in `let`-expressions.

3.4.1 Lambda-Lifting

We could have chosen the easy way: elimination. Recursive functions in `let`-expressions could have been eliminated by lambda-lifting, i.e., globalizing by supplying the free variables as parameters. This is what is done in the latest version of Similix.

Lambda-lifting would unfortunately make specialization degenerate since we don't do specialization with respect to higher-order values. Why? Because sp-functions would have to be globalized and they very often have functions as free variables. Moreover, partial evaluation based on lambda-lifting may actually slow down programs, since specialization in the worst case "does nothing" to the lambda-lifted source program (the result being

```
datatype oneatexp =
  SCON1atexp of scon
| VAR1atexp of string
| CON1atexp of string
| EXCON1atexp of string
| RECORD1atexp of oneexprow Option
| LET1atexp of onedec * oneexp
| PAR1atexp of oneexp

and oneexprow =
  EXPROW1 of lab * oneexp * oneexprow Option

and oneexp =
  ATEXP1exp of oneatexp
| APP1exp of oneexp * oneexp
| TYPED1exp of oneexp * onety
| HANDLE1exp of oneexp * onematch
| RAISE1exp of oneexp
| FN1exp of onematch

and onematch =
  MATCH1 of onemrule * onematch Option

and onemrule =
  MRULE1 of onepat * oneexp
```

Figure 6: Data type for representing code in the generating extension — part 1.

a residual program which is a lambda-lifted version of the source program). This goes against one of our fundamental design decisions: the residual program should not be slower than the original program.

For this reason we would like to "keep scope" by allowing sp-functions to stay in `let`-expressions and allowing the residual functions for a local sp-function to occur inside a corresponding `let`-expression.

3.4.2 Higher-Order Values

Having the residual functions end-up in local scope is not possible in general if specializing with respect to higher-order values. This is seen from the following example. Consider the following declaration

```
fun f d =
  let fun h d' = d + d'
  in (g (fn y => h y) d) + d
  end
and g k d = k d
```

We shall for simplicity assume that `f` and `g` are marked for specialization (even though they are not sp-functions). Let us assume that `d` is dynamic and for a moment that we do specialize with respect to higher-order values. If we keep scope as described above we get the following residual program

```
and onedec =
  VAL1dec of onevalbind
| TYPE1dec of onetypbind
| DATATYPE1dec of onedatbind
| ABSTYPE1dec of onedatbind * onedec
| EXCEPTION1dec of oneexbind
| LOCAL1dec of onedec * onedec
| SEQ1dec of onedec * onedec
| NONFIX1dec of string list
| EMPTY1dec

and onevalbind =
  PLAIN1valbind of onepat * oneexp * onevalbind Option
| REC1valbind of onevalbind

and onetypbind =
  TYPBIND1 of tyvar list * string * onety * onetypbind Option

and onedatbind =
  DATBIND1 of tyvar list * string * oneconbind * onedatbind Option

and oneconbind =
  CONBIND1 of string * onety Option * oneconbind Option

and oneexbind =
  EXBIND1 of string * onety Option * oneexbind Option
| EXEQUAL1 of string * string * oneexbind Option
```

Figure 7: Data type for representing code in the generating extension — part 2.

```
fun f d =
  let fun h d' = d + d'
  in (g_0 d) + d
  end
and g_0 d = h d
```

which suffers from the unfortunate problem that `g_0` refers to `h` which is not in scope. It is relatively easy to see that the problem in the example is specialization with respect to higher-order values as it is through a closure that a variable is transferred out of scope.

For other reasons we have already chosen to specialize with respect to equality-typed values only, so we would deem `h` completely dynamic and not specialize `g` with respect to `h`. That would get a residual program equal to the original program, and the scope problem would then have disappeared. This holds in general, because when one does not specialize with respect to higher-order values, there is no way to transfer function variables out of scope.

3.4.3 Copies of Residual Functions

Still, there are problems with code duplication when keeping scope, even when one does not specialize with respect to higher-order values. Consider specialization of the following function `f`, where we again assume that `f` and `g` are marked for specialization.

```
and oneatpat =
  WILDCARD1atpat
| SCON1atpat of scon
| VAR1atpat of string
| CON1atpat of string
| EXCON1atpat of string
| RECORD1atpat of onepatrow Option
| PAR1atpat of onepat

and onepatrow =
  DOTDOTDOT1
| PATROW1 of lab * onepat * onepatrow Option

and onepat =
  ATPAT1pat of oneatpat
| CON1pat of string * oneatpat
| EXCON1pat of string * oneatpat
| TYPED1pat of onepat * onety
| LAYERED1pat of string * onety Option * onepat

and onety =
  TYVAR1ty of tyvar
| RECORD1ty of onetyrow Option
| CON1ty of onety list * string
| FN1ty of onety * onety
| PAR1ty of onety

and onetyrow =
  TYROW1 of lab * onety * onetyrow Option
```

Figure 8: Data type for representing code in the generating extension — part 3.

```
fun f s d =
  let fun g s' d' = s' + s' + d'
  in (g 2 d) + s + s + d
  end
```

Assuming `f` is specialized with `s` equal to 3 and 4 we get the following residual program.

```
fun f_3 d =
  let fun g_2 d' = 4 + d'
  in g_2 d + 6 + d
  end
fun f_4 d =
  let fun g_2 d' = 4 + d'
  in g_2 d + 8 + d
  end
```

where we see that `g_2` occurs twice in the residual program. It is easy to see that for any locally declared function `f` that does not depend on a static variable of the enclosing scope and is marked for specialization there is a risk of duplicating the declaration of the specialized versions of `f` in the residual program.

On the other hand it is not easy to predict how big a problem this code duplication is in practice. We choose to keep scope; then experience will show whether the code duplication problem will occur often in practice.

3.4.4 The Generating Extension for Let-Expressions

We now consider how to keep scope for nested declarations. First consider the case of static `let`-expressions, i.e., those that should be unfolded during specialization. Here we can simply copy the `let`-expression to the generating extension.

Example 1 The source expression

```
let val x = 2
in f x
end
```

(where `f` is some unspecified sp-function) can be copied into the generating extension, so that the generating extension will produce the residual call `f_2()` (where `f_2` is the residual version of `f` for the static value `x=2`). □

For dynamic `let`-expressions we somehow need to collect the residual functions that are generated for specialization functions in the `let`-expression, so they can be placed in a residual `let`-expression. We shall therefore associate one code list (code accumulator) for every declaration that can possibly contain a specialization function, as mentioned in Section 3.2. Recall, that a code list is simply an updateable variable holding a declaration.

During specialization (i.e., when running the generating extension) a number of residual functions and residual non-functions (such as `val x = y + z`) are generated. We will require that they all have different names, but it is not possible to place all of them in a mutually recursive value binding due to the syntactic restriction in SML that all expressions bound in such a recursive value binding must be of the form `fn` *match* [MTH90, Section 2.9]. Another solution is to use an accumulator for every value binding in the source program. The latter solution has the nice quality that it automatically groups the residual functions, one group for one source value binding. For a single value binding with n sp-functions the generating extension could then look like this

```
val code = new_code ()
val getcode = fn () => SEQ1dec(getcode(),!code)
val sp1 = fn ...
and ...
and spn = fn ...
```

The variable `code` is bound to a fresh empty code list, and the idea is then that the calls to `emit` (see section 3.2) in `sp1`, ..., `spn` uses the `code` variable as argument, whereas `getcode` is used to obtain the residual code for these sp-functions and the preceding ones; note that the call to `getcode` inside `getcode` is *not* a recursive call, but a call to the preceding `getcode` function! In each scope `getcode` is initially bound as follows

```
val getcode = fn () => EMPTY1dec
```

This way only those residual functions which must be recursive are placed in the same codelist, and the order of residual functions follows the order of each mutual recursive group of generating extensions for sp-functions as we wish.

Now we shall see that this strategy also has the advantage of being well-suited for the case of dynamic `let`-expressions. Consider the following dynamic `let`-expression, and assume `DV` is some unspecified dynamic variable.

```
let val sp = fn s => fn d => (s + s) + d
in (sp 3 DV) + (sp 4 DV)
end
```

We want this to result in the following residual `let`-expression

```
let
  val sp_3 = fn d => 6 + d
  and sp_4 = fn d => 8 + d
in
  (sp_3 DV) + (sp_4 DV)
end
```

and we can obtain that by using the following generating extension

```
let
  val getcode = fn () => EMPTY1dec
  val SeenB4List = ref ([] : (int * string) list)
  val code = new_code ()
  val getcode = fn () => SEQ1dec(getcode(),!code)
  fun sp_gen s d =
    let
      val (seen,name) = seenB4 SeenB4List s
      val _ =
        if seen then
          ()
        else
          emit(code,name,["d"],
               ⌈lift_int (s + s) + d⌉)
    in
      ⌈name d⌉
    end
  val body = ⌈(sp_gen 3 DV) + (sp_gen 4 DV)⌉
in
  ⌈let getcode() in body end⌉
end
```

Note how `getcode` is initially bound to a function which returns an empty declaration, such that when the last `getcode()` is evaluated, only the residual functions generated by `sp_gen` are collected and placed in the residual `let`-expression. Moreover, note the introduction of the variable `body`. We cannot just unfold the binding of `body`, since we have to make sure that the calls to the `sp_gen` functions have been evaluated before `getcode()` is evaluated — otherwise we would not get a hold on the residual functions.

For a `let`-expression that is residual because it binds something residual the situation is almost as above. For instance,

```
let val d' = d + d
        s' = s * s
in d' * d' + s'
end
```

can be handled by

```
let
  val getcode = fn () => EMPTY1dec
  (* SeenB4List not needed *)
  val code = new_code ()
  val getcode = fn () => SEQ1dec(getcode(),!code)
  val d' =
    let
      (* never seen before *)
      val _ =
        emit(code,"d'",[],
               ⌈d + d⌉)
    in
      ⌈d'⌉
    end
  val s' = s * s
  val body = ⌈d' * d' + lift_int s'⌉
in
  ⌈let getcode() in body end⌉
end
```

The only difference is that we need not check whether the specialization has been done before because we always need exactly one residual version of `d'`.

We have now seen how the the idea of code lists and `getcode` functions can be used to keep scope for nested declarations.

3.5 Partially Static Structures

A structured value which contains dynamic values, but where parts of it are known is called a *partially static structure* [Mog88]. The classic example of a partially static structure is a list of elements where the elements are unknown but the structure, i.e., the length of the list, is known. Partially static structures can be split into separate variables before, during or after specialization. The advantages of splitting partially static structures include possibilities for register allocation, and avoidance of runtime indirections, and it is important in order to remove a complete layer of interpretation during specialization.[3]

As remarked in [JGS93] specialization of partially static structures implies *arity raising:* one partially static parameter may give rise to several parameters in the residual function. A residual function generated by specializing a function with respect to a partially static list of known length l, for example, will have l parameters. Arity raising has been investigated by Sestoft [Ses86], Romanenko [Rom90], Mogensen [Mog88], and [Mog89a].

[3] In Section 6.2 we show an example of how our system, partly due to the splitting of partially static structures, has removed a complete layer of interpretation for an interpreter for a small imperative language.

Mogensen [Mog89b] suggested to use a syntactic transformation phase before specialization called *binding-time separation*, which transforms a program that operates on partially static structures to an equivalent one whose data structures are all either fully static or fully dynamic. This approach was used by Anne de Niel [dN93], who termed it *program bifurcation*, in the context of a first-order functional language in order to simplify the specialization phase such that she could achieve self-application of her specializer. As our approach to specialization makes no limitations on the specialization phase — we have the full SML language at our disposal in the generating extensions — we will not use binding-time separation. We choose instead to split during specialization.

3.5.1 Safety and Partially Static Structures

Safe specialization of partially static structures requires some care [Bon92] [JGS93]. Consider specialization of the following expression

```
let val ps = (s,d) in #1 ps end
```

and assume that `s` is fully static and `d` is fully dynamic, so that the pair `ps` is partially static. Simply reducing this expression during specialization to `s` is unsafe as we then discard the dynamic computation `d` which might loop, cause exceptions, perform assignments et cetera. One can also risk computation duplication as can be seen by considering the following expression

```
let val ps = (s,d) in (#2 ps) + (#2 ps) end
```

A simple way to avoid these problems is to introduce a `let`-expression to bind the components before the pair is constructed:

```
let
  val s = s
  and d = d
in
  let val ps = (s,d) in #1 ps end
end
```

Then, using the usual binding-time assignment for `let`-expressions, the outer `let` will be deemed dynamic such that specialization yields the following residual expression.

```
let val d = d in s end
```

This approach has some disadvantages, however. It means that the entire expression will be dynamic, again using the traditional binding-time assignment for `let`-expressions, even though the body expression of the `let`-expression is static; this is unfortunate as it hinders further specialization.

A way to handle this problem is to perform the specialization in continuation passing style as proposed by Bondorf [Bon92] such that the specialization context can propagate through dynamic `let`-expressions. We believe this is an important optimization, but it complicates matters so we choose not to specialize in continuation passing style. When enough experience has been gained with the cogen approach to specialization, it should not be difficult fully to exploit the ideas of [Bon92].

3.5.2 Specialization of Partially Static Structures

From now on we shall assume that `let`-expressions have been inserted as explained in the preceding section, and start considering how to specialize partially static structures. Note that it is a bit more complicated than in existing partial evaluators [Lau89] [Bon92] since our compiler generator must generate code which can perform the actual splitting of partially static structures when the generating extension is executed. We will consider splitting of records and constructed values — these are the only kinds of values (Figure 13 in Definition of Standard ML defines what a value in SML is [MTH90]) it is reasonable to consider to split.

The underlying idea is due to Mogensen [Mog89a, Chapter 6], and is also used by Launchbury [Lau89, Section 6.3] in the context of a small first-order functional language, and Bondorf [Bon93] in the context of Similix.

3.5.3 Partially Static Records

Consider the following example, where we for simplicity shall assume that `f` is marked for specialization.

```
val f = fn ps => (#1 ps) + (#2 ps) + (#3 ps)
...
val x = f (11,22,d1)
```

Assume that the structure of the argument to `f` is known, and that the first and second components of `ps` are static and that the third component of `ps` is dynamic. Essentially the generating extension for this function will be like the following.

```
val SeenB4list = ...
val code = ...
val getcode = ...
val f_gen = fn ps =>
  let
    val (actualsres, gensplitexp, genseenB4, newvars) =  sp_do ps
    val ps = gensplitexp
    val sp_res_exp = mk_sp_res_exp actualsres
    val (seen,name) = seenB4 SeenB4List genseenB4
    val _ =
      if seen then
        ()
      else
        emit(code,name,mk_sp_res_pat newvars,
             ⌈lift_int (#1 ps) + (#2 ps) + (#3 ps)⌉)
  in
    ⌈name sp_res_exp⌉
  end
```

The `sp_do` function performs the actual splitting, so when `f_gen` is applied to the tuple `(11,22,d1)`, we get the following values for `actualres`, `gensplitexp`, `genseenB4`, and `newvars`:

Variable	Value
actualres	[d1]
gensplitexp	(11,22,ATEXP1exp(VAR1atexp "x17")
genseenB4	(11,22,blankcode)
newvars	["x17"]

actualsres (the actual parameters of the residual call) is the list of the dynamic parts of `ps`. In the example this is only `d1`.

gensplitexp (the value that is split in the generating extension) is the tuple with fresh variables inserted for the dynamic parts — `x17` in the example[4]

genseenB4 (the value passed to seenB4 in the generating extension) is the tuple with the dynamic parts blanked out (so that they do not count when seenB4 does its equality tests). The blanking is done by replacing all dynamic parts with a constant piece of code whose actual value is irrelevant. An alternative to this approach would be to use an alternative data type with the dynamic components removed.

newvars is the list of the fresh variables inserted into `gensplitexp`.

The function `mk_sp_res_pat` creates a residual pattern (the argument pattern of the residual function), in this example ⌈(`x17`)⌉. Correspondingly, `mk_sp_res_exp` is used to create the actual argument of the call to the residual function, here `d1`. The re-binding of `ps` to gensplitexp ensures that when a dynamic component is selected in the body of `f`, as `#3 ps`, then the result becomes the code consisting of the fresh variable inserted for the dynamic component, which is given a value when the residual program is run.

The residual program obtained when evaluating this generating extension (i.e., when `f_gen` is applied to `(11,22,d1)`) is

```
val f_res = fn x17 => 33 + x17
```

where we see that only the dynamic parts of the original partially static structure is to be supplied at residual time.

Now how should we define the `sp_do` function? We would like a simple clean way of doing it, so we without too much trouble are able to *generate* such `sp_do` functions. For the example case under consideration we will use the following function.

[4]Recall that dynamic expressions (code) during the evaluation of the generating extension is represented by the data type `oneexp`, hence the variable is converted to an expression as shown.

```
val rec sp_do =
  fn (p : (int * int * oneexp)) =>
  let
    val (actualsres_1,gensplitexp_1,genseenB4_1,newvars_1) =
         sp_do_12 (#1 p)
    val (actualsres_2,gensplitexp_2,genseenB4_2,newvars_2) =
         sp_do_12 (#2 p)
    val (actualsres_3,gensplitexp_3,genseenB4_3,newvars_3) =
         sp_do_3 (#3 p)
  in
    (actualsres_1 @ actualsres_2 @ actualsres_3,
     (gensplitexp_1,gensplitexp_2,gensplitexp_3),
     (genseenB4_1,genseenB4_2,genseenB4_3),
     newvars_1 @ newvars_2 @ newvars_3)
  end
and sp_do_12 =
  fn (p : int) => ([],p,p,[])
and sp_do_3 =
  fn (p : oneexp) =>
  let
    val newvar = fresh_var ()   (* returns a fresh variable *)
  in
    ([p],ATEXP1exp(VAR1atexp newvar),blankcode,[newvar])
  end
```

The things to notice is the uniformity of the `sp_do`-functions, and that `sp_do_12` can be applied for both the first and second component in `sp_do`. We need only one `sp_do` function for each combination of SML type and binding time, except for the case of binding time "dynamic": here we can use the same `sp_do` function, no matter what the SML type is, as the value is represented by code (of type `oneexp`).

We have now seen the approach we shall use for specializing partially static records. In the next subsection we consider partially static data types and later on pattern matching where we will refine the ideas; in the following chapter the refined ideas will be made precise.

3.5.4 Partially Static Data Types

A partially static data type is simply a data type for which the values are partially static. Recall that a value of a data type in SML is either a constructor or a constructor paired with another value. As we have chosen to use monovariant binding times, we will distinguish between exactly two situations: either the constructors of a data type are all known during specialization or the constructors are not known during specialization. This simple observation is important: it holds for all kinds of data types, including mutually recursive data types. For an example, consider the following recursive data type

```
datatype tree =
  LEAF
| NODE of int * tree * tree
```

Values of this data type can either be fully dynamic (constructors not known) or fully static (constructors and elements known) or partially static (constructors known, elements in nodes unknown).

If the constructors of a data type t are not known we will require that the argument of a constructed t value is unknown too. This restriction can be lifted by considering constructor specialization [Mog93], which we will briefly discuss in Section 7.2. Hence if the constructors are not known we have fully dynamic values, which present no problems.

If the constructors are known during specialization, we can in fact treat data types in the same way we treat records. For the data type `tree`, for example, we can define an `sp_do` function in the following straightforward manner, assuming that the elements of the nodes are unknown.

```
val rec sp_do_tree =
  fn LEAF => ([],LEAF,LEAF,[])
   | (NODE x) =>
       let
         val (actualsres_x, gensplitexp_x,genseenB4_x,newvars_x) =
           sp_do_x x
       in
         (actualsres_x,
          NODE gensplitexp_x,
          NODE genseenB4_x,
          newvars_x)
       end
and sp_do_x =
  fn (p: (oneexp * tree * tree)) =>
  let
    val (actualsres_1,gensplitexp_1,genseenB4_1,newvars_1) =
         sp_do_elem (#1 p)
    val (actualsres_2,gensplitexp_2,genseenB4_2,newvars_2) =
         sp_do_tree (#2 p)
    val (actualsres_3,gensplitexp_3,genseenB4_3,newvars_3) =
         sp_do_tree (#3 p)
  in
    (actualsres_1 @ actualsres_2 @ actualsres_3,
     (gensplitexp_1,gensplitexp_2,gensplitexp_3),
     (geenseenB4_1,geenseenB4_2,geenseenB4_3),
     newvars_1 @ newvars_2 @ newvars_3)
  end
and sp_do_elem =
  fn (p : oneexp) => .... (* as sp_do_12 above *)
```

Note that this requires that the following data type declaration is present in the generating extension.

```
datatype tree = LEAF | NODE of oneexp * tree * tree
```

The idea is that the parts of the type-expressions that correspond to dynamic values are exchanged with the code type `oneexp`. In the next chapter we define precisely how this is to be done.

3.6 Pattern Matching

In this section we study specialization of patterns. We shall see that the idea of hand-writing a compiler generator directly instead of writing a self-applicable partial evaluator makes it possible to obtain very good results: to the best of our knowledge this is the first `cogen` (hand-written or generated by self-application) that handles as complicated patterns as the SML patterns directly. First we review the problems with making a self-applicable partial evaluator for languages with pattern matching, and then go on to describe how we specialize patterns using the cogen approach. To simplify the exposition we defer discussion of pattern matching on exception constructors to Section 3.9.[5]

3.6.1 Pattern Matching and Self-Applicable Partial Evaluation

Parts of this section is based on [Bon90, Chapter 6] where Anders Bondorf discusses partial evaluation of Tree, a language with pattern matching as in SML.

One can consider a self-applicable partial evaluator as a smart self-interpreter, as the partial evaluator has to contain an interpreter in order to evaluate fully static expressions. Indeed Jones, Gomard, and Sestoft, in [JGS93, Section 7.4], argue that to obtain a reasonable self-applicable partial evaluator, one must be able to specialize a self-interpreter, so the optimality criterion is satisfied. The optimality criterion is: specializing a self-interpreter with respect to a program `p` should yield a residual program `p'` which is almost identical to `p`, except for trivial differences. Therefore it is reasonable to simplify the discussion and consider specialization of a self-interpreter `sint` for SML.

Consider the following expression (modified for SML from an expression in [Bon90, Chapter 6]) containing relatively simple pattern matching

```
let
  val f = fn (A, B) => 1
           | (A, C) => 2
in
  f (x, y)
end
```

where `A`, `B`, and `C` are constructors of the same data type, and `x` and `y` are variables, which we shall assume are static. Suppose we have a traditional partial evaluator `mix`, and assume further that `sint` is written naïvely following the dynamic semantics of SML [MTH90, Section 6].[6] Specializing `sint` with respect to this expression then yields something like the following (the result was obtained by "hand-specializing" using the rules of the Definition of SML as the self-interpreter)

[5]Anticipating events we can say that under certain conditions which we will establish, exception constructors behave essentially as other constructors.

[6]The inference rules for the dynamic semantics for the Core Language [MTH90, Section 6] are deterministic and can be conceived of as an interpreter.

```
let val v' =
  let val VE = let val VE = if x = A then emptyVEmap else FAIL
               in
                 if VE = FAIL then FAIL
                 else VE ++ if y = B then emptyVEmap else FAIL
               end
  in
    if VE = FAIL then FAIL else eval_scon(1)
  end
in
  if v' = FAIL then
    let VE = let val VE = if x = A then emptyVEmap else FAIL
             in
               if VE = FAIL then FAIL
               else VE ++ if y = B then emptyVEmap else FAIL
             end
    in
      if VE = FAIL then FAIL
      else 2
    end
  else v'
end
```

where the infix operator `++` here is addition of finite maps, the implementation of the semantic operation used in [MTH90]. The constant `emptyVEmap` is the neutral element of that operator. Optimality has certainly not been achieved; the naïve if-then-else structure has been inherited from the simple pattern matching algorithm in the self-interpreter; in essence, redundant match rules have been generated. As argued by Bondorf, the number of redundant rules depend on the product of the depth of the patterns for every match rule, which is completely unacceptable for realistic programs. A first thought is, of course, that one can try to remove the redundant rules, but this is complex and Bondorf reports that experience shows that unacceptably slow partial evaluation is the result. This led Bondorf to give up partial evaluation of a language with pattern matching; instead he considered partial evaluation of a simple tree decision language, into which a language with pattern matching can be translated.

3.6.2 Pattern Matching and Cogen

We regard a (sub-)pattern as static (or compile-time) if all the values it is matched against are static, or if the pattern consists of a variable (which can also match dynamic values). This is formalized via a two-level language in the next chapter. We remark that a pattern in itself is always known during specialization — it is in the program and does not depend on any input (just like constants) — which is the reason for the reference to the values the pattern will be matched against.

In the preceding subsection, we tried to hand-specialize a self-interpreter in order to consider self-application. Self-application is not an issue in the compiler generator setting, so in the examples we shall consider ordinary SML programs with pattern matching. Note the line of argument: the goal is to be able to specialize programs written in a language with pattern matching, and to be able to generate compilers from interpreters by partial evaluation; to achieve the latter goal one can make a self-applicable partial evaluator, but as we have seen this impedes good specialization of pattern matching if the partial evaluator uses pattern matching. Another way to achieve the goal of producing compilers is to write a compiler generator, where there is no need to consider specializing a self-interpreter, and we shall see that this approach yields good specialization of pattern matching as well.

Pattern Matching and Decomposition of Values

Pattern matching in SML, besides being used to choose among several branches, is used to decompose values. For instance, selecting a component out of a record is done by pattern matching: the derived form `#`*lab* is short for `fn {`*lab* `= x, ...} => x`. In this section we consider how to treat this aspect of pattern matching. Assume the source program contains the following two data type declarations (there are no further problems with recursive data types),

```
datatype t1 = C1 of int * int | D1 of int
datatype t2 = C2 of t1 * int | D2 of int * t1 * int
```

and assume furthermore that the constructors of `t1` are unknown and the constructors of `t2` are known, and that the integer components of `C2` and `D2` are static. For the expression

```
fn (D2(i,C1(a,b),j)) => (i + j) + a
```

it should then be possible to perform some decomposition, corresponding to matching on the constructor `D2` and on the record argument of `D2`. Moreover, as `i` and `j` are static we should be able to perform the addition of `i` and `j` at compile-time. This can be achieved by generating the following generating extension, where `d` is a fresh variable.

```
fn (D2(i,d,j)) =>
  ⌈(fn (C1(a,b)) => lift_int (i + j) + a) d ⌉
```

The trick is to insert fresh variables for the dynamic parts of the pattern, so as much decomposing of values as possible is performed during specialization, and only defer the pattern matching which cannot be performed at compile-time to residual time. Note that the variable `d` is used to bind a dynamic value so an expression performing the matching in the residual program can be generated as shown.

Pattern Matching and Choice Between Several Branches

Consider the following function, where we use the data types from the previous subsection, and make the same assumptions about binding times as before.

```
val sp = fn (C2(C1(a,b),i)) => i
          | (D2(i,C1(a,b),j)) => (i + j) + b
          | (D2(i,D1 a,j)) => (i + j) + a
          | x => 3
```

This is an example of an sp-function, where one cannot determine which branch to choose at compile-time (since the constructors of the data type `t1` are not known). The last branch is only used to make the match exhaustive. Assume we specialize `sp` with respect to the partially static value `D2(5,d,7)` for some dynamic value `d`. Then one of the last three branches will be chosen, depending on which value we will get for `d` at residual time; notice that the first branch certainly cannot be chosen. We would like to obtain a residual function like the following.

```
val sp_res =
  fn (C1(a,b)) => 12 + b
   | (D1 a) => 12 + a
   | x => 3
```

Note that rule `x => 3` is in fact redundant. This is because the dynamic patterns happen to be exhaustive in this example; if this had not been the case, the rule would not have been redundant. We will not attempt to check whether the dynamic patterns are exhaustive. We will never generate more residual match rules than in the original program, and any redundant rules can easily be removed by post-processing, so this is no problem.

The residual program can be obtained by generating a generating extension like the following.

```
val SeenB4list = ...
val code = ...
val sp_gen = fn p =>
  let
    val (actualsres, gensplitexp, genseenB4, newvars) =
      sp_do p
    val sp_res_exp = mk_sp_res_exp actualsres
    val (seen,name) = seenB4 SeenB4List genseenB4
    val _ =
      if seen then
        ()
      else
        emit(code,name,
          mk_sp_res_fn
          [Some (⌈(C1(a,b)) =>
                 (fn (C2(_,i)) => lift_int i) gensplitexp⌉)
           handle Match => None,

           Some (⌈(C1(a,b)) =>
                 (fn (D2(i,_,j)) => ⌈lift_int (i + j) + b⌉) gensplitexp⌉)
           handle Match => None,

           Some (⌈(D1 a)) =>
                 (fn (D2(i,_,j)) => ⌈lift_int (i + j) + a⌉) gensplitexp⌉)
           handle Match => None,

           Some (⌈x =>
                 (fn _ => lift_int 3) gensplitexp⌉)
           handle Match => None])
  in
    ⌈name sp_res_exp⌉
  end
```

When this generating extension is executed we get the above shown residual program. The function `mk_sp_res_fn` takes a list of optional residual match rules and generates a residual `fn` *match* expression for the present match rules. The idea is as follows. We attempt, for each of the original match rules to generate a residual match rule, where the pattern is the dynamic part of the original pattern, and the expression is the result of specializing the right hand side of the original rule. The attempt can fail, whereby the exception `Match` is raised, if we on the basis of the static available information can determine that this branch can not be chosen. This is the case for the first rule: we try to match on the static part of the pattern, the dynamic parts are "blanked" by inserting wildcard patterns, and as the value of `gensplitexp` in the example under consideration is `D2(5,ATEXPexp(VARatexp "x18",7)` the matching fails, and the first element of the list argument to `mk_sp_res_fn` becomes `None`.

Notice how the matching also binds the static pattern variables, making it possible to exploit their values when specializing the expressions of the match rules. For instance, `i` and `j` are bound to `5`, resp. `7` in the case of the second rule. The dynamic pattern variables, e.g., `b` in the case of the second rule, are not bound to any values; they take their values at residual time. This is the reason for the backquotes around `b`.

So this way we are able to remove those branches at compile-time that on the basis of static information cannot be chosen and, moreover, pattern matching in the residual program will only be made on the dynamic parts of the original pattern. Another nice thing about this approach is that we do not have to code up a pattern matcher, we simply exploit the pattern matching in SML.

We would like the reader to notice that static pattern variables can be bound to objects of different type. Consider the following match, for example,

```
fn C(1, ..., x, ...) => ... x ...
 | C(_, ..., (a,b,c), ...) => ... a ... b ... c ...
```

where we assume that the first component of the record argument of `C` is unknown, such that we cannot determine which branch to choose at compile-time. If the record patterns `x` and `(a,b,c)` are static, then the variable `x` can be bound at compile-time to a record, whereas the variables `a`, `b` and `c` can be bound to the components of the record without any trouble.

Let us now consider how the above strategy fits with our strategy for partially static structures, where the `sp_do` functions insert fresh variables for the dynamic parts, and these fresh variables become parameters of the residual functions. If we blindly followed that strategy we would not get the dynamic patterns, e.g., `C1(a,b)` from above, placed in the residual program. What we shall do therefore is to associate positions with dynamic parts of a pattern and the corresponding values, and then at specialization time replace those fresh variables that correspond to dynamic patterns with the dynamic patterns. This requires an example, so consider again the data types

```
datatype t1 = C1 of int * int | D1 of int
datatype t2 = C2 of t1 * int | D2 of int * t1 * int
```

and assume now that the constructors of `t1` are not known, that the constructors of `t2` are known, that all integer components of `t2` except the last are known. If we specialize the sp-function

```
val sp = fn (C2(C1(a,b),i)) => i
          | (D2(i,C1(a,b),j)) => i + j
          | (D2(i,D1 a,j)) => (i + 2) + a
          | x => 3
```

with respect to the value `D2(5,d1,d2)` for some dynamic values `d1` and `d2`, we would like to obtain the following residual program.

```
val sp_res =
  fn (C1(a,b), x17) => 5 + x17
   | (D1 a, x17) => 7 + a
   | x => 3
```

where `x17` is a fresh variable generated for the last dynamic component of `D2`. This can be achieved using the following generating extension.

```
val SeenB4list = ...
val code = ...
val sp_gen = fn p =>
  let
    val (actualsres, gensplitexp, genseenB4, newvars) =
      sp_do p "0"
    val sp_res_exp = mk_sp_res_exp actualsres
    val (seen,name) = seenB4 SeenB4List genseenB4
    val _ =
      if seen then
        ()
      else
        emit(code,name,
          mk_sp_res_fn
          [Some (⌈mk_sp_res_pat newvars [(⌈C1(a,b)⌉,"01")] =>
                 (fn (C2(_,i)) => lift_int i) gensplitexp⌉)
           handle Match => None,

           Some (⌈mk_sp_res_pat newvars [(⌈C1(a,b)⌉,"02")] =>
                 (fn (D2(i,_,j)) => ⌈lift_int i + b ⌉) gensplitexp⌉)
           handle Match => None,

           Some (⌈mk_sp_res_pat newvars [(⌈D1 a⌉, "02")] =>
                 (fn (D2(i,_,j)) => ⌈lift_int (i + 2) + a⌉) gensplitexp⌉)
           handle Match => None,

           Some (⌈mk_sp_res_pat newvars [] =>
                 (fn _ => lift_int 3) gensplitexp⌉)
           handle Match => None])
  in
    ⌈name sp_res_exp⌉
  end
```

The idea is that the `sp_do` function enumerates the fresh variables for the dynamic parts after some protocol. We use a simple strategy and employ strings to encode the positions, and let every sp_do function have an argument describing the position of the value which the sp_do function is to treat. The mk_sp_res_pat function then takes a list of dynamic patterns and positions (these positions can be determined at compiler generation time) and replaces the variables in newvars according to the positions. In the next chapter we describe precisely how this generating extension is generated.

Pattern Matching and References

In Standard ML pointers can be dereferenced by pattern matching as in the function

```
fun four (ref 4) = true
  | four _       = false
```

that tests whether an integer pointer is a reference to the integer four. While this hidden dereferencing seems like a convenient feature of the language it causes[7] severe problem for partial evaluation. Recall that the constructor ref has the very unique property that each time it is applied the result is different from anything else, so (ref 4 = ref 4) evaluates to false. In other words: constructed reference values cannot be rebuilt from their components.

Consider now the cocktail of dereferencing by pattern matching, layered patterns (such as "abc as (a,b,c)"), and protection from code duplication by insertion of let-expressions as in the following example

```
fun trouble (p as (s,ref d)) =
  (!(#2 p)) + (!(#2 p)) + s + s + d + d
```

To avoid code duplication we want to insert let-expressions just after = which is quite easy for s and d, but certainly not for p. We must do something and we cannot just insert let val p = p in ... end because that leads to code duplication. We must somehow bind p to the protected versions of s and d if we want to avoid duplication, but we cannot do that either because the value of p is different from the value of (s,ref d).

We have decided to let all layered patterns be residual and not to rebind variables introduced by layered patterns. This way we preserve semantics but get poor specialization results if layered patterns are used. We could handle layered patterns without ref reasonably, but the result would be the same as if we required a preprocessing phase to eliminate these layered patterns.

3.7 Side Effects

Standard ML includes two facilities involving side effects: references and input/output. It is fundamental to the design of our compiler generator that we do not try to perform any of the operations dealing with these constructs. In a language like Standard ML this is not prohibitive to partial evaluation as it is not in the spirit of the language to use assignments excessively. Other languages, like C, require that the partial evaluator can

[7] It can be argued which of several features that is the real culprit, but this one seems like the most bizarre to us.

handle side effects — otherwise no specialization takes place. In principle some of the ideas developed in [And92] and [And93] might have been used to perform some of the assignments of a program at specialization time, but we have not researched this in any detail.

For input/output the situation is a little different: even if we were able to decide that the program should output something we might not want it done at specialization time; this argument of course fails to hold for temporary files but in general we want the specialization to be passive.

For these elaborate reasons and excuses this section will describe how all side effects can be postponed to run time, or as some put it "deported to the residual program." The same decision was made for Similix [BD91].

3.7.1 Preservation of Semantics

Partial evaluation should preserve the semantics of programs. However obvious this may sound it has some restraining consequences on what the partial evaluator is allowed to do.

First of all one imperative operation of the source program must correspond to exactly one imperative action in the residual program; calculations may neither be duplicated nor thrown away. For example we cannot throw the calculation of the second component away in

```
fun dup f s d =
  let
    p = (s,f d)
  in
    (#1) p
  end
```

Second, evaluation order must be preserved as for example the expressions `i := !i + 1` and `i := !i * 2` are non-interchangeable. There are other reasons for enforcing these two points: duplication is inefficient and exceptions are evaluation order dependent. This is covered in more detail in sections 3.1.5 and 3.9.

3.7.2 Postponing Actions

Postponing all input/output actions is simple: declare the built-in stream-handling functions (`open_in`, `close_in`, `input`, `lookahead`, `end_of_stream`, `open_out`, `close_out`, and `output`) to have the binding-time type `D`. This will ensure that all calls to these functions are made residual.

The assignment facilities are the functions `:=` and `!` plus the constructor `ref` that belongs to an unnamed data type[8] with that constructor only. These three pervasives are polymorphic but it turns out that this causes no problem as long as we simply postpone all actions. The actual postponing is done by giving the three pervasives and the unnamed data type the binding-time type `D`. This makes all applications of these pervasives residual.

[8] Classifying it as a data type justifies the fact that dereferencing can be done by pattern matching.

We must also arrange for uses of the above variables (i.e., all identifiers but `ref`) plus `std_in` and `std_out` to get unchanged through to the residual program. The easiest way to do this is to have all occurrences double-underlined. Double underlinings (which have nothing to do with the dotted underlining) will be explained in Section 4.2. Right now the important thing is that it can be done.

3.8 Miscellaneous

3.8.1 Pervasives

The standard functions of Standard ML, the so-called pervasives, need special attention because they cannot be treated monovariantly if we want any calculations at all done at specialization time; for instance, a single dynamic equality test cannot be allowed to force all equality tests to be dynamic. Similar special treatment is given by Similix which has special syntactic phrases for applications of primitive operations.

We cannot use the operator names like "`+`" and "`div`" in the generating extension for both the static and the dynamic cases so we reserve them for the static cases and use a construction like `(mk_op "+")` in the dynamic cases where `mk_op` is some suitable function available to the generating extension.

3.8.2 Overloading

Certain of Standard ML's pervasives are overloaded. For instance there is a function called `+` that adds integers and at the same time a function also called `+` that adds reals. The language definition specifies that the type of every occurrence must be determined from the context, but it does not specify exactly what context.

Since partial evaluation means changing a lot of context and partial evaluation via compiler generation even means separating static (compile time) context from dynamic (residual time) context, there is a fair chance that the context that determined the type of an overloaded occurrence in the source program no longer does that neither in the generating extension nor in the residual program. Example:

```
fun overload x =
  if true then
    x+x
  else
    0
```

In this example the context clearly shows the addition to be integer addition. After partial evaluation that context will just as clearly disappear.

Since sufficient type information is present at cogen-time the solution is to provide that information explicitly in both the generating extension and the residual program. The easiest way to do this is simply to generate type ascriptions to give the types of the overloaded operators explicitly.

3.9 Exceptions

In this section we discuss how to specialize programs with exceptions. As we do not expect the reader to remember all the details of the dynamic semantics of exceptions in Standard ML we first describe the semantics. Readers confident with SML exceptions may skip the first section. Thereafter we turn to partial evaluation of exceptions.

3.9.1 Generativity

The standard function `hd` that extracts the first element of a list might have been written this way:

```
exception Hd
fun hd [] = raise Hd
  | hd (x::xs) = x
```

This program defines a new exception, `Hd`, with no argument. Defining an exception means that the constructor is added to the special predefined data type `exn`. Values of this type can also be passed as parameters, bound to variables, and are subject to pattern matching as values of any other data type. It is not an equality type. Unlike other data types however values can be "raised" meaning that the value is converted to a so-called exception package that immediately starts propagating itself out to the nearest, dynamically seen, (if any) `handle`-phrase that matches without any normal evaluation going on in the meantime. For instance this means that if the evaluation of the expression `f(x,y,g x,h y)` causes an exception to be raised while evaluating `g x` then that exception package will be the final result and `h y` will not be computed.

Certain exceptions are predefined by Standard ML, and implementations are likely to define more. We summarize the predefined exceptions in this table:

Exception	Reason
`Neg`, `Sum`, ...	(Arithmetic) errors in the corresponding function.
`Io` of string	Error during input/output.
Interrupt	User has requested the calculation to be interrupted.
Match	The value used for pattern matching against a `fn` match (not a `handle` match) does not match any of the match rules.
Bind	The value in a value binding does not match the pattern it was supposed to match.

Recall from Section 2.1 that the programs we treat are required not to raise exception `Match` or `Bind` unless explicitly done by a `raise`-construct. This also makes it possible for us to require that the user program does not use the identifier `Match` but something else, so that there are no collisions with the generating extension's use of the built-in `Match` exception. Furthermore we will not consider the nondeterministic exception `Interrupt` and the `Io` exception will never be raised except explicitly as all input/output is deferred to the residual program as described in Section 3.7; that makes `Io` just like a user defined exception.

The constructors of the `exn` data type are almost like constructors of any other data type apart from the way they are defined. They are however subject to generativity: the above mentioned extension of the `exn` data type is done at runtime. If the extension is done locally, different constructors are defined each time the declaration is executed. This program prints the value 1:

```
fun strange n =
  let
    exception Zero
  in
    if n=0 then
      raise Zero
    else
      (strange (n-1)) handle Zero => 0
  end

val it = (strange 2) handle _ => 1
```

as the versions of `Zero` used for pattern matching ($\texttt{Zero}_2$ and $\texttt{Zero}_1$) are not the same as the one that was raised ($\texttt{Zero}_0$). As programs with local exceptions can be difficult to understand some authors recommend avoiding local exceptions whenever possible, see for example [Pau91, pages 115–116].

3.9.2 Exceptions and Partial Evaluation

As exceptions are normally used only to detect and handle error conditions implying that a `raise`-construction will only be reached when some kind of an error occurs, it might be reasonable to consider the easy solution: exporting (aka deporting) all exceptions packages to the residual program by treating `raise` as a function with binding time $\texttt{D} \rightarrow \texttt{D}$ and assuming that there are no other sources of exceptions, i.e., standard exceptions are not raised. The last assumption is not unreasonable as we have already eliminated the important ones, see above.

It turns out that simple deportation is not possible as it ruins the binding-time properties. Consider the `hd` example in the previous section. A binding-time analysis usually assigns the same binding time to different branches of a match, so the result of any call to `hd` would therefore be dynamic if `raise` always returned a dynamic result. Even worse: we might know (or assume) that the function was never called with an empty list and might then choose to return some dummy, but static, value in that case. Thus dirty programming improves the binding-time separation. We do not like that and it does not solve the problem when the exceptions are actually used.

We conclude from the above considerations that we must be prepared to handle exceptions at specialization time. Specifically we must handle exceptions that are raised due to the overstrict nature of partial evaluation. By overstrict we mean that all branches of a dynamic match will be specialized instead of just one being evaluated.

In the core of Standard ML matches are the only non-strict constructions. Example:

```
if d then
  1 div 0
else
  0
```

The generating extension of this expression will evaluate both branches of the dynamic conditional so we cannot just say

```
⌈if d then
  lift_int (1 div 0)
else
  lift_int 0⌉
```

as this will result in an exception package and not a piece of code as we would like. In fact we must be able to perform the equivalent of lifting a static value. In this example it is clear that only the `Div` exception can be raised and only in the first branch but in general we must be prepared for any exception:

```
⌈if d then
  (lift_int (1 div 0)) handle x => mk_exn x
else
  (lift_int 0) handle x => mk_exn x⌉
```

where `mk_exn` : $exn \rightarrow$ `oneexp` is a function that takes any value of type `exn` and produces a corresponding piece of code to raise it, i.e., in effect lifts the exception. This function must somehow be made available to the generating extension, but for now we will just assume that such a function can be programmed.

The above method of handling exceptions in overstrict contexts has no problems with user defined exceptions, and when the source program raises an exception explicitly the generating extension can do the same. But what about explicit handling of exceptions? The generating extension of an expression may raise the handled exception either at specialization time or at residual time as demonstrated by the division respectively the whole conditional above. The solution is both to handle the exception at specialization time and to generate code to handle it at residual time. In the case of the division there is no residual code involved so we can be sure that no exceptions are generated at residual time, but this optimization opportunity is not easily detected[9] so we refrain from it.

3.9.3 Globalizing Exceptions

In the previous section we assumed that a function `mk_exn` to lift values of type `exn` could be programmed. If we handle arguments to exception constructors in such a way that they can be lifted when needed, the problem reduces to knowing the constructors. For any normal constructed data type this would end the discussion, but remember that the `exn` type has the very special property that any execution of an `exception` declaration adds a constructor to the type. Thus we do not even know the number of constructors

[9] We might have used the fact that our binding-time analysis will never assign `S` as the binding time of an expression that involves code in any way. This condition would however cease to hold if we improved the specialization by doing it in continuation passing style as done in [Bon92].

and they need not have different names! We believe this to be impossible to handle so we require all exceptions to be declared at top level in which case they're uniquely identified by their constructor names.

In this section we show how a Standard ML program with locally declared exceptions can be transformed into an equivalent one with only globally declared exceptions if the program satisfies certain conditions. The existence of this algorithm ensure that we have *not* limited the class of programs that our methods can handle. The globalization can be considered further simplification of the kind discussed in section 2.2 albeit a simplification requiring much rewriting.

We will now use the assumptions that the program is alpha-converted, monomorphic, that all data type declarations precede all value bindings in the program, that matches are exhaustive, and that all type declarations have been eliminated.

Global exceptions are first moved so they reside between the data types and the value bindings. For syntactic reasons and without loss of generality we also assume that all exceptions have a parameter, the unit value if nothing else, and that exception aliases (aliases introduced by `exception E = E'`) have been eliminated.

The globalization algorithm is syntax-directed but hard to describe in the core syntax so we use a slightly higher syntactic level. The idea is to mimic the semantic definition (see [MTH90, equations 106, 138, 146, and 156]) by introducing version numbers[10] of local exceptions explicitly. For a locally defined exception `E` we perform the following transformation:

Construct	Action
`exception E of` τ	Insert global `exception E of` $\text{int} * \tau$ and replace local declaration with `val Eno = mk_newint()`
`E` (as an expression)	Replace by `fn x => E (Eno,x)`

`... E pat ... => exp`	Replace by
	`a as (...E(n,pat)...) =>`
	`  if n=Eno then`
	`    exp`
	`  else (case a of <rest cases of match>)`

Important points:

- The identifiers `Eno`, `a`, `n`, and `x` are assumed not to be defined elsewhere. The function `mk_newint` is a function that returns a different integer each time it is called.
- The above translations will fail if matches are not exhaustive.
- Global exceptions are required to be monomorphic in Standard ML. The globalized exceptions will be monomorphic because the whole program is.
- The pattern translation rule can lead to quite a lot of code if many exception constructors are matched. A smarter translation scheme could have specified that all exception numbers be checked by the same `if` expression.

[10]These numbers are called exceptions names in [MTH90].

- After globalization the semantic differences between normal constructors and exception constructors have all disappeared. We will therefore treat exception constructors just like normal constructors. A consequence of this is that if just one dynamic exception (a dynamic value of type `exn`) is raised then all exceptions will be dynamic!

- We may hereafter assume or equivalently obtain by rewriting that all `handle`-expressions have exactly one match rule (for which the pattern is a variable). We will then always know which branch to choose and that simplifies specialization a bit as we do not have to consider specialization point insertion at `handle`-expressions.

- The rewriting rules generate layered patterns which we always deport to the residual program. This means that if the translation rules are followed blindly all exceptions will be deported to the residual program if pattern matching is performed of any of the globalized exceptions (remember that exceptions declared by the user globally are treated much better). To get a better result it is necessary to 1) create an analysis to identify local exceptions for which generativity is not exploited so that these can be globalized by simple moving, and 2) eliminate as many of the `as`-clauses generated by the above algorithm as possible; it could be done by recreating the value on the right-hand side if the pattern did not contain `ref` (because rebuilding using `ref` would change the semantics of the program). Such analyses are outside the scope of this thesis.

The example from earlier becomes (after adding a unit parameter to the exception constructor and making the first match exhaustive)

```
exception Zero of (int*unit);
fun strange n =
  let
    val zero_no = mk_newint()
  in
    if n=0 then
      raise (fn x => Zero(zero_no,x)) ()
    else
      ((strange (n-1)) handle
         a as (Zero(n,())) =>
           (if n=zero_no then
              0
            else
              case a of x => raise x)
        | x => raise x)
  end;

(strange 2) handle _ => 1
```

Admittedly it is not a pretty sight but then again programs output by programs seldom are. If would have been a little nicer if we had not added the unit parameter to `Zero` but then it would not have shown all the principles. Note that eliminating layered patterns is not a problem here: simply replace `a` with `Zero(n,())`.

3.10 Summary

We have presented our design decisions in this chapter and described how to specialize the different construct of the Simplified Bare Language.

We have covered termination properties, code duplication, evaluation order dependency, and lots of other items to ensure preservation of semantics.

To obtain high quality generating extensions and residual programs we have also examined call unfolding, partially static data structures, arity raising, exceptions, pattern matching, and side effects.

For readability we have presented the problems and our solutions in groups. We have deliberately not told the full story and how to fit all the pieces together — this is done in the following chapter.

Chapter 4

Two-Level Simplified Bare Language Semantics

I shall be sorry to doubt the word of such a wise and inspired man, but his meaning, though probably clear to you, is the reverse of clear to me.

— PLATO, *The Republic*

In this chapter we develop a formal specification of the compiler generator for the Simplified Bare Language. We begin with defining a *two-level* version of the language, in which binding times are explicit in the syntax.

Not all two-level programs are allowed, so we define a static semantics of the two-level language. This static semantics ensures, among other things, that the *congruence* property — no static expression depends on a dynamic value [Jon88] [JGS93] [Lau89] — is satisfied. Programs that obey the static semantics are called well-annotated, since the two-level program is an annotated version of the source program.

After defining the static semantics we define the dynamic semantics for the two-level language. An implementation of the dynamic semantics is the compiler generation proper.

In the next chapter we will describe how to obtain a well-annotated program from a well-typed source program. This will be done by the binding-time analysis which, for now, can be considered a black box. When we include binding-time analysis, the cogen approach can be illustrated as in Figure 9.

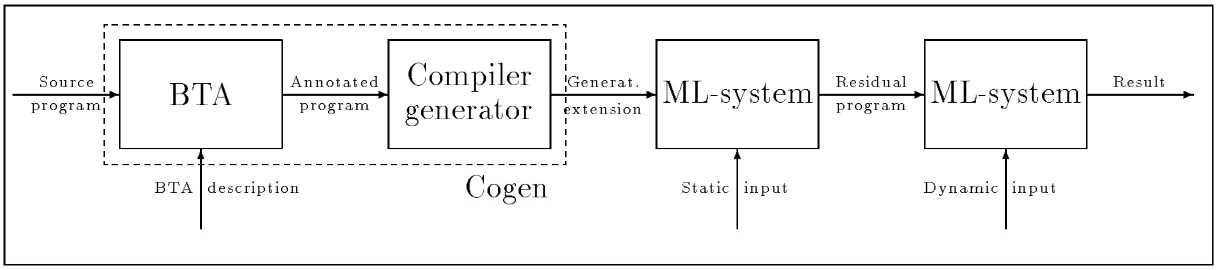

Figure 9: Three-phase evaluation via a generating extension.

4.1 Syntax of Two-Level Simplified Bare Language

In the Two-Level Simplified Bare Language every phrase which can be either static or dynamic appears in two versions: a specialization time version and a residual time version. For example `fn` *2match* denotes a lambda expression to be evaluated at specialization time whereas $\underline{\texttt{fn}}$ *2match* denotes a lambda expression to be evaluated at residual time.

The use of a two-level syntax to express the separation of binding times was founded by the Nielsons [NN88] and has later been used in λ-mix [GJ91] [Gom92], Similix [Bon92], and C-Mix [And92]. This way of communicating the binding-time separation to a specializer and, in our case, a compiler generator is applicable when the binding-time separation is monovariant. However, as we shall discuss in Chapter 7, polyvariance can be achieved using the same principles.

We shall see later that we will not only use the two-level syntax to communicate the binding-time separation to the compiler generator. In the dynamic semantics of the two-level language we shall refer to binding-time information which can be obtained from the static semantics; specifically we will use the binding-time information communicated this way to guide the generation of the so-called `sp_do` functions which are used to extract the static and dynamic parts of arguments. This has the minor drawback that one cannot study our dynamic semantics completely independently of the static semantics (as one for instance can consider the dynamic semantics of SML without having studied the static semantics of SML [MTH90] [MT91]). The reason for this choice is that a two-level syntax expressing the same information will be complicated due to recursive types. We believe our approach is the cleanest way of doing things.

The grammar rules for our Two-Level Simplified Bare Language appear in Figures 10 and 11. We have the same syntactic restrictions as for the Simplified Bare Language, c.f. Section 2.2.

For variables, we in fact have three annotated forms. The first, *var*, is used for variables that will be bound to non-dynamic values at specialization time; the second, $\underline{var}$, is used for variables that will be bound to code at specialization time; and the third, $\underline{\underline{var}}$, is used for variables that will be bound at residual time. The reason for differentiation between the first two is technical and will be explained later. For the rest of the cases of two-level atomic expressions the binding-time separation is obvious. For expression rows the two cases express whether the structure of an expression is known or unknown during specialization. For expressions we have made use of tags, `atexpexp`, resp. $\underline{\texttt{atexpexp}}$, to express whether or not the atomic expression is dynamic or not. It could seem superfluous, but the tags are used in order to generate a correctly typed expression (more about this later). Likewise, tags have been introduced to differentiate between applications to be performed at specialization time (`@`) and applications to be performed at residual time ($\underline{\texttt{@}}$). The tag `lift_exp` signals the presence of a static expression in a dynamic context; we elaborate on the use of this below.

The separation of matches and match rules is the obvious one. If a match has only one match rule we have the problem that there's nothing to underline. Because we neither run across this problem in in the examples nor in the formal rules we give, no confusion should arise. Other grammar productions have the same problem.

For declarations we have only one kind. The reason is simply that the binding-time separation is expressed "at one level below," e.g., a data type declaration can declare several data types, which need not all have the same binding time.

If there is no underlining of the equality sign in a value binding, then the binding should be performed at specialization time. One underline expresses that the value binding should be performed at residual time and two underlines express that a specialization function is bound (to the variable on the left-hand side). Specialization functions were introduced in section 3.1.5.

Underlining in data type bindings and constructor bindings expresses whether or not the constructors of a data type are known during specialization. Likewise for exception bindings.

The underlining of patterns simply expresses whether or not the pattern should occur in the generating extension or in the residual program. The exception is that layered patterns are always underlined, as layered patterns are always put into the residual program as explained in Section 3.6.2.

Underlining in type-expressions is used to express which parts of the argument record of a constructor will be known at residual time.

Example 2 Consider the following version of the Ackermann function where `m` is assumed static and `n` dynamic.

```
val rec ack = fn m => fn n =>
  (fn 0 => n+1
    | m' => (fn 0 => ack (m'-1) 1
              | n' => ack (m'-1) (ack m' (n'-1))) n) m
```

The two-level version, with inserted sp-function is shown below. For readability we have chosen to write the `atexpexp` tags as square brackets, `[` and `]`, instead, possibly underlined. The `atpatpat` tags are not shown. Special constants are shown in the normal syntax for integers and strings.

```
val rec ack =
  fn m =>
     fn n =>
        [(fn 0 => [([mk-op] @ ["+"])] @ [ {1 = [ n ], 2 = lift_exp [1]} ]
          | m' => [([sp] @ [m'])] @ [ n ]
         )
        ] @
        [m]
and sp =
  fn m' =>
     fn 0 =>
        [ack] @ [([-] @ [{1 = [m'], 2 = [1]}])] @ lift_exp [1]
        |
        n' =>
        [ack] @ [([-] @ [{1 = [m'], 2 = [1]}])] @
        [([ack] @ [m'] @
          [([([mk-op] @ ["-"])] @
            [ {1 = [ n' ], 2 = lift_exp [1]} ]
           )
          ]
         )
        ]
```

Note that we used pattern matching directly on the integer instead of going through the equality function and making pattern matching on boolean. This just makes the example smaller. □

Example 3 Below we show the two-level version of the data types `t1` and `t2` from Chapter 3 and the two-level version of the function sp also from Chapter 3. Recall that the constructors of `t1` are unknown while the constructors and the integer components of `t2` are known. We use the same notational conventions as in the preceding example.

```
datatype t1 = C1 of { 1 : int , 2 : int } | D1 of int
datatype t2 = C2 of { 1:t1 , 2:int } | D2 of { 1:int , 2:t1 , 3:int }

val sp = fn (C2 { 1 = C1 { 1 = a , 2 = b } , 2 = i }) =>
              lift_exp [i]
          | (D2 { 1 = i , 2 = C1 { 1 = a , 2 = b } , 3 = j }) =>
              [ ([mk-op] @ ["+"]) ] @ { 1 = lift_exp (i+j) , 2 = b }
          | (D2 { 1 = i , 2 = D1 a , 3 = j }) =>
              [ ([mk-op] @ ["+"]) ] @ { 1 = lift_exp (i+j) , 2 = a }
          | x => lift_exp [3]
```

As it can be seen from this example the two-level syntax is not a convenient way to write examples. □

Use of `lift_exp`

Note the use of lift operations in the Ackermann example above. This lift operator is used when a static expression appears in a dynamic context and directs the compiler generator to generate some code which at specialization time can turn the value v, which the argument expression of `lift_exp` evaluates to at specialization time, into a piece of residual code that evaluates to v at residual time.

We will only allow lifting of basic values during specialization; this restriction is also placed in λ-mix [GJ91] [Gom92], Similix [Bon92] and C-Mix [And92]. The reason is the following. If one allows to lift structured values, one can risk an exponential increase in the amount of memory usage in the residual program compared to the original program.

4.1.1 Annotation-forgetting function

The syntax of the Two-Level Simplified Bare Language has deliberately been defined such that it corresponds nicely with the syntax of the Simplified Bare Language, c.f. Figures 4 and 5 (there is one exception which we explain below). The motivation is that it is then simple to define an annotation-forgetting function [Gom92], which given a phrase of the Two-Level Simplified Bare Language returns a phrase of the Simplified Bare Language. This is important, because it allows us to define, see definition 4, an operator which given a two-level phrase returns the SML type of the corresponding "one-level" phrase.

Definition 1 (Annotation-forgetting function) *The annotation-forgetting function ϕ is the overloaded function which when applied to a two-level phrase 2phrase returns a one-level phrase phrase which differs from 2phrase only in that all annotations (underlines and tags* `@`*,* $\underline{\texttt{@}}$*,* `lift_exp` *et cetera) are removed and parentheses are inserted around the argument in application expressions.*

4.1.2 On the SML-Type of Code

Note in Figure 10 that the parameter in a two-level application expression is of phrase class expression (as opposed to atomic expression as for Standard ML). The reason is technical and could be described as the need for a "stronger type discipline" than Standard ML enforces. In the generating extension we need a type for representing residual code. Previous partial evaluators have just used the expression type, but we have two expression types: atomic and non-atomic. There seems to be no good arguments for preferring either one so we have arbitrarily chosen non-atomic expressions, i.e., type *exp* from Figure 6. Having made this choice we must ensure that whenever a variable in the generating extension is bound to a piece of code, then the variable is of type *exp*. Code can come from two different places: applications or pattern matching. Applications bind atomic expressions while pattern matching on tuples binds non-atomic expressions, so we must do something or else the generating extension will not in general be well typed. By changing the syntax slightly we make sure that dynamic variables always will be bound to values of type *exp* and never to values of type *atexp*.

It should be clear that it would not be a solution, simply to choose *atexp* instead of *exp*. It would move the problems, but not solve them. It is less clear that carefully engineered dynamic semantics might also have been used, but we think that would require three versions of applications instead.

As a surprising side effect of the small change in syntax we also observe that we need not use a lift operation for atomic expressions. Before the change such an operation was needed to make it possible to lift constant arguments to functions, but now the parenthesis-like `atexpexp` production uses atomic expressions and that can be lifted itself.

$$
\begin{array}{lcl}
\mathit{2atexp} & ::= & \mathit{scon} \\
 & & \mathit{var} \\
 & & \underline{\mathit{var}} \\
 & & \underline{\underline{\mathit{var}}} \\
 & & \mathit{con} \\
 & & \underline{\mathit{con}} \\
 & & \mathit{excon} \\
 & & \underline{\mathit{excon}} \\
 & & \mathtt{\{}\langle \mathit{2exprow} \rangle\mathtt{\}} \\
 & & \underline{\mathtt{\{}}\langle \mathit{2exprow} \rangle\underline{\mathtt{\}}} \\
 & & \mathtt{let}\ \mathit{2dec}\ \mathtt{in}\ \mathit{2exp}\ \mathtt{end} \\
 & & \underline{\mathtt{let}}\ \mathit{2dec}\ \underline{\mathtt{in}}\ \mathit{2exp}\ \underline{\mathtt{end}} \\
 & & \mathtt{(}\ \mathit{2exp}\ \mathtt{)} \\
 & & \underline{\mathtt{(}}\ \mathit{2exp}\ \underline{\mathtt{)}} \\
\mathit{2exprow} & ::= & \mathit{lab}\ \mathtt{=}\ \mathit{2exp}\ \langle\ \mathtt{,}\ \mathit{2exprow}\rangle \\
 & & \mathit{lab}\ \underline{\mathtt{=}}\ \mathit{2exp}\ \langle\ \underline{\mathtt{,}}\ \mathit{2exprow}\rangle \\
\mathit{2exp} & ::= & \mathtt{atexpexp}\ \mathit{2atexp} \\
 & & \underline{\mathtt{atexpexp}}\ \mathit{2atexp} \\
 & & \mathit{2exp}_1\ \mathtt{@}\ \mathit{2exp}_2 \\
 & & \mathit{2exp}_1\ \underline{\mathtt{@}}\ \mathit{2exp}_2 \\
 & & \mathtt{fn}\ \mathit{2match} \\
 & & \underline{\mathtt{fn}}\ \mathit{2match} \\
 & & \mathit{2exp}\ \mathtt{handle}\ \mathit{2match} \\
 & & \mathit{2exp}\ \underline{\mathtt{handle}}\ \mathit{2match} \\
 & & \mathtt{raise}\ \mathit{2exp} \\
 & & \underline{\mathtt{raise}}\ \mathit{2exp} \\
 & & \mathtt{lift\_exp}\ \mathit{2exp} \\
\mathit{2match} & ::= & \mathit{2mrule}\ \langle\ \mathtt{|}\ \mathit{2match}\rangle \\
 & & \mathit{2mrule}\ \langle\ \underline{\mathtt{|}}\ \mathit{2match}\rangle \\
\mathit{2mrule} & ::= & \mathit{2pat}\ \mathtt{=>}\ \mathit{2exp} \\
 & & \mathit{2pat}\ \underline{\mathtt{=>}}\ \mathit{2exp} \\
\mathit{2dec} & ::= & \mathtt{val}\ \mathit{2valbind} \\
 & & \mathtt{datatype}\ \mathit{2datbind} \\
 & & \mathtt{exception}\ \mathit{2exbind} \\
 & & \\
 & & \mathit{2dec}_1\ \langle;\rangle\ \mathit{2dec}_2 \\
\mathit{2valbind} & ::= & \mathit{2pat}\ \mathtt{=}\ \mathit{2exp}\ \langle\mathtt{and}\ \mathit{2valbind}\rangle \\
 & & \mathit{2pat}\ \underline{\mathtt{=}}\ \mathit{2exp}\ \langle\mathtt{and}\ \mathit{2valbind}\rangle \\
 & & \mathit{2pat}\ \underline{\underline{\mathtt{=}}}\ \mathit{2exp}\ \langle\mathtt{and}\ \mathit{2valbind}\rangle \\
 & & \mathtt{rec}\ \mathit{2valbind}
\end{array}
$$

Figure 10: Syntax of two-level expressions, matches, and declarations

$$
\begin{array}{lll}
\mathit{2datbind} & ::= & \mathit{tycon}\ \texttt{=}\ \mathit{2conbind}\ \langle \texttt{and}\ \mathit{2datbind}\rangle \\
 & & \mathit{tycon}\ \underline{\texttt{=}}\ \mathit{2conbind}\ \langle \texttt{and}\ \mathit{2datbind}\rangle \\
\mathit{2conbind} & ::= & \mathit{con}\ \langle \texttt{of}\ \mathit{2ty}\rangle\ \langle\ \texttt{|}\ \mathit{2conbind}\rangle \\
 & & \underline{\mathit{con}}\ \langle \underline{\texttt{of}}\ \mathit{2ty}\rangle\ \langle\ \underline{\texttt{|}}\ \mathit{2conbind}\rangle \\
\mathit{2exbind} & ::= & \mathit{excon}\ \langle \texttt{of}\ \mathit{2ty}\rangle\ \langle\langle \texttt{and}\ \mathit{2exbind}\rangle\rangle \\
 & & \underline{\mathit{excon}}\ \langle \underline{\texttt{of}}\ \mathit{2ty}\rangle\ \langle\langle \texttt{and}\ \mathit{2exbind}\rangle\rangle \\
\mathit{2atpat} & ::= & \mathit{scon} \\
 & & \underline{\mathit{scon}} \\
 & & \mathit{var} \\
 & & \underline{\mathit{var}} \\
 & & \mathit{con} \\
 & & \underline{\mathit{con}} \\
 & & \mathit{excon} \\
 & & \underline{\mathit{excon}} \\
 & & \texttt{\{}\langle \mathit{2patrow}\rangle\texttt{\}} \\
 & & \underline{\texttt{\{}}\langle \mathit{2patrow}\rangle\underline{\texttt{\}}} \\
 & & \texttt{(}\ \mathit{2pat}\ \texttt{)} \\
 & & \underline{\texttt{(}}\ \mathit{2pat}\ \underline{\texttt{)}} \\
\mathit{2patrow} & ::= & \mathit{lab}\ \texttt{=}\ \mathit{2pat}\ \langle\ \texttt{,}\ \mathit{2patrow}\rangle \\
 & & \mathit{lab}\ \underline{\texttt{=}}\ \mathit{2pat}\ \langle\ \underline{\texttt{,}}\ \mathit{2patrow}\rangle \\
\mathit{2pat} & ::= & \texttt{atpatpat}\ \mathit{2atpat} \\
 & & \underline{\texttt{atpatpat}}\ \mathit{2atpat} \\
 & & \mathit{con}\ \mathit{2atpat} \\
 & & \underline{\mathit{con}}\ \mathit{2atpat} \\
 & & \mathit{excon}\ \mathit{2atpat} \\
 & & \underline{\mathit{excon}}\ \mathit{2atpat} \\
 & & \mathit{var}\ \underline{\texttt{as}}\ \mathit{2pat} \\
\mathit{2ty} & ::= & \texttt{\{}\langle \mathit{2tyrow}\rangle\texttt{\}} \\
 & & \underline{\texttt{\{}}\langle \mathit{2tyrow}\rangle\underline{\texttt{\}}} \\
 & & \mathit{tycon} \\
 & & \underline{\mathit{tycon}} \\
 & & \mathit{2ty}\ \texttt{->}\ \mathit{2ty}' \\
 & & \mathit{2ty}\ \underline{\texttt{->}}\ \mathit{2ty}' \\
 & & \texttt{(}\ \mathit{2ty}\ \texttt{)} \\
 & & \underline{\texttt{(}}\ \mathit{2ty}\ \underline{\texttt{)}} \\
\mathit{2tyrow} & ::= & \mathit{lab}\ \texttt{:}\ \mathit{2ty}\ \langle\ \texttt{,}\ \mathit{2tyrow}\rangle \\
 & & \mathit{lab}\ \underline{\texttt{:}}\ \mathit{2ty}\ \langle\ \underline{\texttt{,}}\ \mathit{2tyrow}\rangle
\end{array}
$$

Figure 11: Syntax of two-level bindings, patterns, and type expressions

4.2 Static Semantics for Two-Level Simplified Bare Language

As already remarked the two-level syntax does not express all the necessary requirements for a sound binding-time separation. In this section we therefore define a static semantics for the Two-Level Simplified Bare Language. A Two-Level Simplified Bare Language program satisfying the static semantics is called *well-annotated*, and the idea is, as in [GJ91], that for any well-annotated program, neither compiler generation nor specialization should "go wrong." We will return to this correctness issue in Section 4.4.

We will regard the binding time of an expression as the type of the expression in the Two-Level Language. Hence we define the static semantics by a two-level type-system. This way well-annotatedness corresponds to well-typedness.

In the next chapter we consider *how* for a given Simplified Bare Language program to construct a Two-Level Simplified Bare Language program satisfying the static semantics given in this section.

Our style of presentation will resemble the style used in the Definition of Standard ML [MTH90], and unless otherwise stated we use the same notation as in [MTH90] (see in particular Sections 4.2 and 4.3 of [MTH90]).

4.2.1 Semantic Objects

In Figure 12 we define the semantic objects of the static semantics. The semantic objects

$$
\begin{array}{rcl@{\quad}c@{\quad}l}
\tau & \in & \mathrm{BType} & = & \{\mathtt{S},\mathtt{D}\} \cup \mathrm{BRecType} \\
 & & & & \cup \mathrm{BConType} \cup \mathrm{BFunType} \\
\varrho & \in & \mathrm{BRecType} & = & \mathrm{Lab} \xrightarrow{\mathrm{fin}} \mathrm{BType} \\
 & & \mathrm{BConType} & = & \mathrm{BTime} \\
\tau \to \tau' & \in & \mathrm{BFunType} & = & \mathrm{BType} \times \mathrm{BType} \\
rc & \in & \mathrm{BTime} & = & \{\mathtt{R},\mathtt{C}\} \\
TE & \in & \mathrm{TyEnv} & = & \mathrm{TyCon} \xrightarrow{\mathrm{fin}} \mathrm{BTime} \\
VE & \in & \mathrm{VarEnv} & = & \mathrm{Var} \xrightarrow{\mathrm{fin}} (\mathrm{BType} \times \mathrm{BTime}) \\
CE & \in & \mathrm{ConEnv} & = & \mathrm{Con} \xrightarrow{\mathrm{fin}} \mathrm{BType} \times \mathrm{BTime} \cup \mathrm{BTime} \\
EE & \in & \mathrm{ExConEnv} & = & \mathrm{ExCon} \xrightarrow{\mathrm{fin}} \mathrm{BType} \times \mathrm{BTime} \cup \mathrm{BTime} \\
E \text{ or } (TE, VE, CE, EE) & \in & \mathrm{Env} & = & \mathrm{TyEnv} \times \mathrm{VarEnv} \times \mathrm{ConEnv} \times \mathrm{ExConEnv}
\end{array}
$$

Figure 12: Semantic Objects

BFunType and BRecType are used for functions, respectively records, whose structure is known at specialization time. A constructed value for which the constructor is known is given binding time `C` whereas a constructed value with unknown constructor is given binding time `D`. The binding-time type `D` will be used for dynamic values, and `S` is used for static values of base type. BTime is (besides being used in the types for constructed values) used as a flag; it is essentially used to signal when some phrase is to be evaluated: `R` is used for runtime (residual-time) and `C` is used for compile-time (specialization time). We will explain the later why and how this is used.

The type environment, TyEnv, is used to tell for each type-constructor in the domain whether the constructors of the data type corresponding to the type-constructor are known.

The variable environment, VarEnv, gives for each variable in the domain its binding-time type and a binding time which is `C` if the variable will be bound at specialization time and `R` if the variable is to be bound at residual time. A variable may be bound at specialization time to something dynamic, e.g., when the identity function is applied to a dynamic value at specialization time (applying a function at specialization time corresponds to unfolding the function); this is the reason why it is not sufficient to bind variables to binding-time types. In the type-system used for λ-mix [Gom92] variables are only bound to binding-time types and that is sufficient in λ-mix because *all* variables are bound at specialization time in a specialization time environment *including* the dynamic variables which are bound to fresh variables to occur in the residual program. In our case we will not have a specialization time environment which we ourselves will manipulate — instead we use the SML environment — and hence we differentiate as described (alternatively we could have merged the phrase classes two-level atomic expressions and two-level expressions, but we prefer not to do this as already explained.)

The constructor environment, ConEnv, tells for each constructor in its domain whether the constructor is known or not during specialization, and if the constructor takes an argument the binding time of this argument. Likewise for the exception constructor environment, ExConEnv.

For environments, we use the following notational convention (almost[1] as in [MTH90]): when we apply an environment (which in reality is a tuple) we use the syntactic class of the argument to determine the relevant function. For instance, $E(con)$ means $CE(con)$ provided that $E = (TE, VE, CE, EE)$ for suitable values of TE, VE, CE, and EE. For such component extraction we use the notation CE of E. Another notational convention is that we usually omit injections and projections on the semantic objects; this should cause no confusion at all.

4.2.2 Initial Static Environment

Just as there is an initial static environment giving types to the pervasives in the static semantics of SML [MTH90, Appendix C], we can define an initial static environment giving binding times for the pervasives.

It is easy to define such a static initial environment E_p, so we will spare the reader the details. For example using the same ideas as in [MTH90, Appendix C], the variable `+` stands for two functions (one to be applied at specialization time, and one to be applied at residual time) as described in Section 3.8.1, so `+` can be conceived of as bound to the following two binding-time types and times in the initial variable environment (the

[1] Here constructors are not bound in the variable environment as in the Definition of SML.

context will always determine which one to use)

$$\{1 \mapsto \mathtt{S}, 2 \mapsto \mathtt{S}\} \to \mathtt{S} \qquad \text{and} \qquad \mathtt{D}$$

We shall assume given an *initial division* describing the binding times of the parameters of the main function. An initial division is simply a variable environment VE_0, giving the binding time for the main function.

4.2.3 Well-Annotatedness of Two-Level Programs

In the next section we define and explain a bunch of inference rules and some semantic operations which are used to define well-annotatedness. Each of the inference rules allows inference of judgements of the form $A \vdash \mathit{phrase} \Rightarrow A'$ where A and A' are semantic objects. Such a judgement can be read as: in context A phrase *phrase* elaborates to A'. For example, $E \vdash \mathit{2dec}_{tl} \Rightarrow (E', rc)$ informally means that in environment E the two-level declaration *2dec* elaborates to environment E' (giving binding times for the declared variables et cetera) and residual/compile-time *rc*.

Definition 2 (Well-Annotated Two-Level Program) *A two-level program $\mathit{2dec}_{tl}$ is well-annotated if $E_p + VE_0 \vdash \mathit{2dec}_{tl} \Rightarrow (E', \mathtt{R})$ can be derived from the inference rules in the following section for some environment E' and binding time* R.

4.2.4 Inference Rules

Notation

In the inference rules we sometimes use a wildcard (written "_") to express that it does not matter what a value is. Moreover, we allow ourselves to use, e.g., *2dec* both as a variable ranging over the two-level declarations and as the phrase class of all two-level declarations (instead we could have defined phrase classes as in [MTH90, Figure 2], but it should cause no confusion).

Two-Level Atomic Expressions

The inference rules for two-level atomic expressions occur in Figure 13. The rules are explained as follows. A constant is always static and is evaluated at specialization time. We use $\underline{\underline{\mathit{var}}}$ for variables that will be bound at residual time, c.f. Section 4.2.1, whereas $\underline{\mathit{var}}$ and *var* are used for variables that will be bound at specialization time. The reason we have two kinds of variables to be bound at specialization time is a bit technical and will be explained in Section 4.3. Variables are given types according to the environment, and here we see a use of both binding-time types and binding-time times (R or C).

Constructors are given types according to the type environment; if the constructor is known (rule 4.5) and has an argument it is to be conceived of as a function; hence the type. Notice that all the constructed values are given type C, no matter if the constructors take an argument or not – we do not have to differentiate between constructed values having different SML types as we are not type-checking in the usual sense.

Dynamic, `D`, record expressions are residual. Parentheses are underlined if the expression within is underlined; note that parentheses are allowed around a dynamic expression (for instance a function application which is performed at specialization time and returns a dynamic value) — so here we use the `C`/`R`-flags to differentiate.

`let`-expressions are residual if the declaration contains a specialization function or binds a dynamic value (expressed by the `R`-flag). A specialization time `let`-expression takes the binding time after the body.

Two-Level Expression Rows

The inference rules for two-level expression rows are in Figure 14. The rules are straightforward; the only thing to notice is that the information of whether or not the expression row is residual is propagated from record atomic expressions, see rules 4.9–4.10.

Two-Level Expressions

The inference rules for two-level expressions are in Figure 15.

The `atexpexp` tag is underlined if the atomic expression below is residual, and the expression takes the binding time of the atomic expression. This tag is used to determine when, at specialization time or at residual time, an atomic expression is to be converted into an expression.

An application expression is compile-time if the function is known, i.e., if the function has a functional type; we can, of course, not apply a dynamic function at specialization time.

In rule 4.23 one is allowed to choose any type τ, so a `raise` expression has "arbitrary" type — just as in the static semantic for Standard ML (see rule 13) in [MTH90]).

The rules for lambda expressions are straightforward.

In rule 4.27 the `lift_exp` operator is used to convert a static expression to a dynamic expression.

Two-Level Matches

The inference rules for two-level matches are in Figure 16. The rules are straightforward.

Two-Level Match Rules

The inference rules for two-level match rules are in Figure 17.

The inferred variable environment for the pattern contains the binding time assumptions made for the variables in the pattern. The easiest way to read this rule is that if the context requires the match rule being residual (dynamic), then the pattern must be residual too, c.f. rule 4.31.

Two-Level Atomic Patterns

The inference rules for two-level atomic patterns are in Figure 18.

Two-Level Atomic Expressions $\boxed{E \vdash \mathit{2atexp} \Rightarrow (\tau, rc)}$

$$\frac{}{E \vdash \mathit{scon} \Rightarrow (\mathtt{S}, \mathtt{C})} \tag{4.1}$$

$$\frac{E(\mathit{var}) = (\tau, \mathtt{C}) \qquad \tau \neq \mathtt{D}}{E \vdash \mathit{var} \Rightarrow (\tau, \mathtt{C})} \tag{4.2}$$

$$\frac{E(\mathit{var}) = (\mathtt{D}, \mathtt{C})}{E \vdash \underline{\mathit{var}} \Rightarrow (\mathtt{D}, \mathtt{C})} \tag{4.3}$$

$$\frac{E(\mathit{var}) = (\mathtt{D}, \mathtt{R})}{E \vdash \underline{\underline{\mathit{var}}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.4}$$

$$\frac{E(\mathit{con}) = (\tau, \mathtt{C})}{E \vdash \mathit{con} \Rightarrow (\tau \rightarrow \mathtt{C}, \mathtt{C})} \qquad \frac{E(\mathit{con}) = \mathtt{C}}{E \vdash \mathit{con} \Rightarrow (\mathtt{C}, \mathtt{C})} \tag{4.5}$$

$$\frac{E(\mathit{con}) = \mathtt{R}}{E \vdash \underline{\mathit{con}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.6}$$

$$\frac{E(\mathit{excon}) = \mathtt{C}}{E \vdash \mathit{excon} \Rightarrow (\tau \rightarrow \mathtt{C}, \mathtt{C})} \qquad \frac{E(\mathit{excon}) = \mathtt{C}}{E \vdash \mathit{excon} \Rightarrow (\mathtt{C}, \mathtt{C})} \tag{4.7}$$

$$\frac{E(\mathit{excon}) = \mathtt{R}}{E \vdash \underline{\mathit{excon}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.8}$$

$$\frac{\langle E \vdash \mathit{2exprow} \Rightarrow (\varrho, \mathtt{C})\rangle}{E \vdash \mathtt{\{}\langle \mathit{2exprow}\rangle\mathtt{\}} \Rightarrow (\{\}\langle +\varrho\rangle, \mathtt{C})} \tag{4.9}$$

$$\frac{\langle E \vdash \mathit{2exprow} \Rightarrow (\mathtt{D}, \mathtt{R})\rangle}{E \vdash \underline{\mathtt{\{}}\langle \mathit{2exprow}\rangle\underline{\mathtt{\}}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.10}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\tau, \mathtt{C})}{E \vdash \mathtt{(}\ \mathit{2exp}\ \mathtt{)} \Rightarrow (\tau, \mathtt{C})} \tag{4.11}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathtt{(}}\ \mathit{2exp}\ \underline{\mathtt{)}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.12}$$

$$\frac{E \vdash \mathit{2dec} \Rightarrow (E', \mathtt{C}) \qquad E + E' \vdash \mathit{2exp} \Rightarrow (\tau, \_)}{E \vdash \mathtt{let}\ \mathit{2dec}\ \mathtt{in}\ \mathit{2exp}\ \mathtt{end} \Rightarrow (\tau, \mathtt{C})} \tag{4.13}$$

$$\frac{E \vdash \mathit{2dec} \Rightarrow (E', \mathtt{R}) \qquad E + E' \vdash \mathit{2exp} \Rightarrow (\mathtt{D}, \_)}{E \vdash \underline{\mathtt{let}}\ \mathit{2dec}\ \underline{\mathtt{in}}\ \mathit{2exp}\ \underline{\mathtt{end}} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.14}$$

Figure 13: Inference Rules for Two-Level atomic expressions

Two-Level Expression Rows $\boxed{E \vdash \mathit{2exprow} \Rightarrow (\tau, \mathit{rc})}$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\tau, \_) \qquad \langle E \vdash \mathit{2exprow} \Rightarrow (\varrho, \mathtt{C})\rangle}{E \vdash \mathit{lab} \texttt{ = } \mathit{2exp} \ \langle \ \texttt{,} \ \mathit{2exprow}\rangle \Rightarrow (\{\mathit{lab} \mapsto \tau\}\langle + \varrho\rangle, \mathtt{C})} \tag{4.15}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{D}, \_) \qquad \langle E \vdash \mathit{2exprow} \Rightarrow (\mathtt{D}, \mathtt{R})\rangle}{E \vdash \mathit{lab} \ \underline{\texttt{=}} \ \mathit{2exp} \ \langle \ \underline{\texttt{,}} \ \mathit{2exprow}\rangle \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.16}$$

Figure 14: Inference Rules for Two-Level expression rows

Underlining is used for residual atomic patterns (for which the third component inferred is R). In rule 4.34 one is allowed to assume any binding-time type τ for the variable.[2] The context will always ensure that only one choice makes sense; through the application rule (4.19) and the rules for value bindings (rules 4.58–4.60).

Two-Level Pattern Rows

The inference rules for two-level pattern rows are in Figure 19.

A pattern row is residual if the record pattern it is part of is, and if the pattern row is residual then so are all the subpatterns.

Two-Level Patterns

The inference rules for two-level patterns are in Figure 20.

The tag for atomic patterns is underlined if the pattern is residual.

Consider rule 4.48, which is applicable if the constructor is known during specialization. We give an example to show how the binding time of an unary constructor is communicated through the environment. Consider the following specialization time two-level static application expression, where A is assumed to be a constructor taking an integer as argument,

```
(fn A x => atexpexp x) @ (A atexpexp 1)
```

Assume E is an environment for which $E(\mathtt{A}) = (\mathtt{S}, \mathtt{C})$. Then the inference tree shown below proves the judgement $E \vdash$ `(fn A x => atexpexp x) @ (A atexpexp 1)` $\Rightarrow (\mathtt{S}, \mathtt{C})$, and, informally speaking, thereby shows the well-annotatedness of the expression. Let $*$ be the following inference tree.

[2]This is like the perhaps more familiar case of a lambda abstraction, where one is usually allowed to assume any type for the abstracted variable when type checking the body of the lambda expression.

Two-Level Expressions $\boxed{E \vdash \mathit{2exp} \Rightarrow (\tau, \mathit{rc})}$

$$\frac{E \vdash \mathit{2atexp} \Rightarrow (\tau, \mathtt{C})}{E \vdash \mathtt{atexpexp}\ \mathit{2atexp} \Rightarrow (\tau, \mathtt{C})} \tag{4.17}$$

$$\frac{E \vdash \mathit{2atexp} \Rightarrow (\mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathtt{atexpexp}}\ \mathit{2atexp} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.18}$$

$$\frac{E \vdash \mathit{2exp}_1 \Rightarrow (\tau \to \tau', \mathtt{C}) \qquad E \vdash \mathit{2exp}_2 \Rightarrow (\tau, \_)}{E \vdash \mathit{2exp}_1\ \mathtt{@}\ \mathit{2exp}_2 \Rightarrow (\tau', \mathtt{C})} \tag{4.19}$$

$$\frac{E \vdash \mathit{2exp}_1 \Rightarrow (\mathtt{D}, \_) \qquad E \vdash \mathit{2exp}_2 \Rightarrow (\mathtt{D}, \_)}{E \vdash \mathit{2exp}_1\ \underline{\mathtt{@}}\ \mathit{2exp}_2 \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.20}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\tau, \mathtt{C}) \qquad E \vdash \mathit{2match} \Rightarrow \mathtt{C} \to \tau}{E \vdash \mathit{2exp}\ \mathtt{handle}\ \mathit{2match} \Rightarrow (\tau, \mathtt{C})} \tag{4.21}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{D}, \_) \qquad E \vdash \mathit{2match} \Rightarrow \mathtt{D}}{E \vdash \mathit{2exp}\ \underline{\mathtt{handle}}\ \mathit{2match} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.22}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{C}, \mathtt{C})}{E \vdash \mathtt{raise}\ \mathit{2exp} \Rightarrow (\tau, \mathtt{C})} \tag{4.23}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathtt{raise}}\ \mathit{2exp} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.24}$$

$$\frac{E \vdash \mathit{2match} \Rightarrow \tau \to \tau'}{E \vdash \mathtt{fn}\ \mathit{2match} \Rightarrow (\tau \to \tau', \mathtt{C})} \tag{4.25}$$

$$\frac{E \vdash \mathit{2match} \Rightarrow \mathtt{D}}{E \vdash \underline{\mathtt{fn}}\ \mathit{2match} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.26}$$

$$\frac{E \vdash \mathit{2exp} \Rightarrow (\mathtt{S}, \mathtt{C})}{E \vdash \mathtt{lift\_exp}\ \mathit{2exp} \Rightarrow (\mathtt{D}, \mathtt{R})} \tag{4.27}$$

Figure 15: Inference Rules for Two-Level Expressions

Two-Level Matches $\boxed{E \vdash \mathit{2match} \Rightarrow \tau}$

$$\frac{E \vdash \mathit{2mrule} \Rightarrow \tau \rightarrow \tau' \qquad \langle E \vdash \mathit{2match} \Rightarrow \tau \rightarrow \tau' \rangle}{E \vdash \mathit{2mrule}\ \langle\ \texttt{|}\ \mathit{2match}\rangle \Rightarrow \tau \rightarrow \tau'} \tag{4.28}$$

$$\frac{E \vdash \mathit{2mrule} \Rightarrow \texttt{D} \qquad \langle E \vdash \mathit{2match} \Rightarrow \texttt{D} \rangle}{E \vdash \mathit{2mrule}\ \langle\ \underline{\texttt{|}}\ \mathit{2match}\rangle \Rightarrow \texttt{D}} \tag{4.29}$$

Figure 16: Inference Rules for Two-Level matches

Two-Level Match Rules $\boxed{E \vdash \mathit{2mrule} \Rightarrow \tau}$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \texttt{C}) \qquad E + \mathit{VE} \vdash \mathit{2exp} \Rightarrow (\tau', \_)}{E \vdash \mathit{2pat}\ \texttt{=>}\ \mathit{2exp} \Rightarrow \tau \rightarrow \tau'} \tag{4.30}$$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \texttt{D}, \texttt{R}) \qquad E + \mathit{VE} \vdash \mathit{2exp} \Rightarrow (\texttt{D}, \_)}{E \vdash \mathit{2pat}\ \underline{\texttt{=>}}\ \mathit{2exp} \Rightarrow \texttt{D}} \tag{4.31}$$

Figure 17: Inference Rules for Two-Level match rules

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{E(\texttt{A}) = (\texttt{S},\texttt{C})}{E \vdash \texttt{A} \Rightarrow (\texttt{S},\texttt{C})} \quad \cfrac{}{E \vdash \texttt{x} \Rightarrow (\{\texttt{x} \mapsto (\texttt{S},\texttt{C})\}, \texttt{S}, \texttt{C})}}{E \vdash \texttt{A x} \Rightarrow (\{\texttt{x} \mapsto (\texttt{S},\texttt{C})\}, \texttt{C}, \texttt{C})} \quad \cfrac{\cfrac{\cfrac{}{E + \{\texttt{x} \mapsto (\texttt{S},\texttt{C})\}(\texttt{x}) = (\texttt{S},\texttt{C})}}{E + \{\texttt{x} \mapsto (\texttt{S},\texttt{C})\} \vdash \texttt{x} \Rightarrow (\texttt{S},\texttt{C})}}{E + \{\texttt{x} \mapsto (\texttt{S},\texttt{C})\} \vdash \texttt{atexpexp x} \Rightarrow (\texttt{S},\texttt{C})}}{\vdots}}{E \vdash \texttt{fn A x => atexpexp x} \Rightarrow (\texttt{C} \rightarrow \texttt{S},\texttt{C})}}{E \vdash \texttt{(fn A x => atexpexp x)} \Rightarrow (\texttt{C} \rightarrow \texttt{S},\texttt{C})}$$

and let $**$ be the following inference tree.

$$\cfrac{\cfrac{E(\texttt{A}) = (\texttt{S},\texttt{C})}{E \vdash \texttt{A} \Rightarrow (\texttt{S} \rightarrow \texttt{C},\texttt{C})} \qquad \cfrac{\cfrac{}{E \vdash \texttt{1} \Rightarrow (\texttt{S},\texttt{C})}}{E \vdash \texttt{atexpexp 1} \Rightarrow (\texttt{S},\texttt{C})}}{E \vdash \texttt{A @ atexpexp 1} \Rightarrow (\texttt{C},\texttt{C})}$$

Then the complete inference tree is the following.

$$\frac{* \qquad **}{E \vdash \texttt{(fn (A x) => atexpexp x) @ (A @ atexpexp 1)} \Rightarrow (\texttt{S},\texttt{C})}$$

Notice how the binding-time for the constructor must be recorded in the enviroment. It is through use of the environment that the annotation requirements of data type bindings and type expressions are expressed.

Two-Level Declarations

The inference rules for two-level declarations are in Figure 21.

Two-Level Atomic Patterns $\boxed{E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \tau, \mathit{rc})}$

$$\frac{}{E \vdash \mathit{scon} \Rightarrow (\{\}, \mathtt{S}, \mathtt{C})} \tag{4.32}$$

$$\frac{}{E \vdash \underline{\mathit{scon}} \Rightarrow (\{\}, \mathtt{D}, \mathtt{R})} \tag{4.33}$$

$$\frac{}{E \vdash \mathit{var} \Rightarrow (\{\mathit{var} \mapsto (\tau, \mathtt{C})\}, \tau, \mathtt{C})} \tag{4.34}$$

$$\frac{}{E \vdash \underline{\mathit{var}} \Rightarrow (\{\mathit{var} \mapsto (\mathtt{D}, \mathtt{R})\}, \mathtt{D}, \mathtt{R})} \tag{4.35}$$

$$\frac{E(\mathit{con}) = \mathtt{C}}{E \vdash \mathit{con} \Rightarrow (\{\}, \mathtt{C}, \mathtt{C})} \tag{4.36}$$

$$\frac{E(\mathit{con}) = \mathtt{R}}{E \vdash \underline{\mathit{con}} \Rightarrow (\{\}, \mathtt{D}, \mathtt{R})} \tag{4.37}$$

$$\frac{E(\mathit{excon}) = \mathtt{C}}{E \vdash \mathit{excon} \Rightarrow (\{\}, \mathtt{C}, \mathtt{C})} \tag{4.38}$$

$$\frac{E(\mathit{excon}) = \mathtt{R}}{E \vdash \underline{\mathit{excon}} \Rightarrow (\{\}, \mathtt{D}, \mathtt{R})} \tag{4.39}$$

$$\frac{\langle E \vdash \mathit{2patrow} \Rightarrow (\mathit{VE}, \varrho, \mathtt{C})\rangle}{E \vdash \mathtt{\{}\langle \mathit{2patrow}\rangle\mathtt{\}} \Rightarrow (\{\}\langle + \mathit{VE}\rangle, \{\}\langle + \varrho\rangle, \mathtt{C})} \tag{4.40}$$

$$\frac{\langle E \vdash \mathit{2patrow} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})\rangle}{E \vdash \underline{\mathtt{\{}}\langle \mathit{2patrow}\rangle\underline{\mathtt{\}}} \Rightarrow (\{\}\langle + \mathit{VE}\rangle, \mathtt{D}, \mathtt{R})} \tag{4.41}$$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})}{E \vdash \mathtt{(}\ \mathit{2pat}\ \mathtt{)} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})} \tag{4.42}$$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathtt{(}}\ \mathit{2pat}\ \underline{\mathtt{)}} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})} \tag{4.43}$$

Figure 18: Inference Rules for Two-Level atomic patterns

Two-Level Pattern Rows $\boxed{E \vdash \mathit{2patrow} \Rightarrow (\mathit{VE}, \tau, \mathit{rc})}$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C}) \qquad \langle E \vdash \mathit{2patrow} \Rightarrow (\mathit{VE}', \varrho, \mathtt{C})\rangle}{E \vdash \mathit{lab}\ \mathtt{=}\ \mathit{2pat}\ \langle\ \mathtt{,}\ \mathit{2patrow}\rangle \Rightarrow (\mathit{VE}\langle + \mathit{VE}'\rangle, \{\mathit{lab} \mapsto \tau\}\langle + \varrho\rangle, \mathtt{C})} \tag{4.44}$$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R}) \qquad \langle E \vdash \mathit{2patrow} \Rightarrow (\mathit{VE}', \mathtt{D}, \mathtt{R})\rangle}{E \vdash \mathit{lab}\ \underline{\mathtt{=}}\ \mathit{2pat}\ \langle\ \underline{\mathtt{,}}\ \mathit{2patrow}\rangle \Rightarrow (\mathit{VE}\langle + \mathit{VE}'\rangle, \mathtt{D}, \mathtt{R})} \tag{4.45}$$

Figure 19: Inference Rules for Two-Level pattern rows

Two-Level Patterns $\boxed{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \mathit{rc})}$

$$\frac{E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})}{E \vdash \mathtt{atpatpat}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})} \tag{4.46}$$

$$\frac{E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathtt{atpatpat}}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})} \tag{4.47}$$

$$\frac{E(\mathit{con}) = (\tau, \mathtt{C}) \qquad E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})}{E \vdash \mathit{con}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{C}, \mathtt{C})} \tag{4.48}$$

$$\frac{E(\mathit{con}) = \mathtt{R} \qquad E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathit{con}}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})} \tag{4.49}$$

$$\frac{E(\mathit{excon}) = (\tau, \mathtt{C}) \qquad E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \tau, \mathtt{C})}{E \vdash \mathit{excon}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{C}, \mathtt{C})} \tag{4.50}$$

$$\frac{E(\mathit{excon}) = \mathtt{R} \qquad E \vdash \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})}{E \vdash \underline{\mathit{excon}}\ \mathit{2atpat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})} \tag{4.51}$$

$$\frac{E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \mathtt{D}, \mathtt{R})}{E \vdash \mathit{var}\ \underline{\mathtt{as}}\ \mathit{2pat} \Rightarrow (\{\mathit{var} \mapsto (\mathtt{D}, \mathtt{R})\} + \mathit{VE}, \mathtt{D}, \mathtt{R})} \tag{4.52}$$

Figure 20: Inference Rules for Two-Level patterns

The binding-time component inferred is used to tell whether or not something of residual time is declared in the declaration. The semantic operation $\sqcup$ used in rule 4.57 and defined below is simply used to express that a sequential declaration declares something residual if one of the declarations does. We use the normal least-upper-bound symbol as one could look at the two-point set $\{\mathtt{C}, \mathtt{R}\}$ as a lattice with $\mathtt{C} \sqsubset \mathtt{R}$.

Definition 3 (The $\sqcup$ operator) *The compile-time/residual-time combination operator* $\sqcup : \mathrm{BTime} \times \mathrm{BTime} \to \mathrm{BTime}$ *operator is defined by:* $\mathtt{R} \sqcup \mathtt{R} = \mathtt{R}$, $\mathtt{R} \sqcup \mathtt{C} = \mathtt{R}$, $\mathtt{C} \sqcup \mathtt{R} = \mathtt{R}$, *and* $\mathtt{C} \sqcup \mathtt{C} = \mathtt{C}$.

The weird looking rule 4.56 is correct; it is for the empty declaration which is always compile-time.

Note that we do not infer a binding-time flag for data type and exception bindings since by the syntactic restrictions they always occur in the outermost declaration, which *always* declares something residual, the main function if nothing else.

Two-Level Declarations $\boxed{E \vdash \mathit{2dec} \Rightarrow (E', \mathit{rc})}$

$$\frac{E \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})}{E \vdash \mathtt{val}\ \mathit{2valbind} \Rightarrow (E', \mathit{rc})} \tag{4.53}$$

$$\frac{E \vdash \mathit{2datbind} \Rightarrow (\mathit{TE}, \mathit{CE})}{E \vdash \mathtt{datatype}\ \mathit{2datbind} \Rightarrow ((\mathit{TE}, \{\}, \mathit{CE}, \{\}), \mathtt{R})} \tag{4.54}$$

$$\frac{E \vdash \mathit{2exbind} \Rightarrow \mathit{EE}}{E \vdash \mathtt{exception}\ \mathit{2exbind} \Rightarrow ((\{\}, \{\}, \{\}, \mathit{EE}), \mathtt{R})} \tag{4.55}$$

$$\frac{}{E \vdash \qquad \Rightarrow (\{\}, \mathtt{C})} \tag{4.56}$$

$$\frac{E \vdash \mathit{2dec}_1 \Rightarrow (E_1, \mathit{rc}_1) \qquad E + E_1 \vdash \mathit{2dec}_2 \Rightarrow (E_2, \mathit{rc}_2)}{E \vdash \mathit{2dec}_1\ \langle ; \rangle\ \mathit{2dec}_2 \Rightarrow (E_1 + E_2, \mathit{rc}_1 \sqcup \mathit{rc}_2)} \tag{4.57}$$

Figure 21: Inference Rules for Two-Level declarations

Two-Level Value Bindings

The inference rules for two-level value bindings are in Figure 22. Recall that by the syntactic restrictions the *2pat* in the two-level value bindings in rules 4.58–4.60 is always a variable.

The semantic predicate sp is defined below. It is used to determine whether or not the value binding is a binding of an sp-function. If the expression *2exp* is not dynamic and not an sp-function, then it is simply a specialization time binding (rule 4.58). If the expression is residual and not an sp-function it is a residual binding. This is used for the binding of dynamic values in the inserted `let`-expressions. Finally, rule 4.60 expresses the requirements for an sp-function.

The semantic operation TypeOf used in the antecedent of rule 4.60 is defined as follows (recall our notational conventions from page 72).

Definition 4 (The TypeOf **operator)** *Let* Type *be the semantic object defined in the Definition of Standard ML, [MTH90, Figure 10]. For a given two-level program 2dec, let* $dec = \phi(2dec)$*. For any 2exp or 2match in 2dec, the operator* $\mathrm{TypeOf} : 2exp \cup 2match \to \mathrm{Type}$ *yields the SML type of the corresponding* $exp = \phi(2exp)$ *and* $match = \phi(2match)$ *given by the elaboration of the dec in the initial static basis ([MTH90, Appendix C]) according to the Static Semantics of SML [MTH90, Section 4].*

A two-level sp-function has n arguments, where $n \geq 1$. For all the arguments it must be the case that either they are dynamic or they are of SML equality type[3] as we only specialize with respect to equality-typed values, c.f. Section 3.1.4. The sp predicate is defined as follows.

Definition 5 (The sp **predicate)** *The predicate* $\mathrm{sp}(2exp, E, \tau)$ *is satisfied iff* $2exp =$ `fn` var_1 `=>` $\cdots$ `=>` `fn` var_n `=>` $\underline{\texttt{fn}}$ *2match for some* $var_1,\ldots,var_n$ *and some 2match or*

1. $2exp =$ `fn` var_1 `=>`$\cdots$`fn` var_n`=>fn` $2pat_1$ `=>` $2exp_1 \mid \cdots \mid 2pat_m$ `=>` $2exp_m$, $m \geq 2$ *for some* $var_1, \ldots, var_n$, $2pat_1, \ldots, 2pat_m$, *and* $2exp_1, \ldots, 2exp_m$ *and*
2. $\tau = \tau_1 \to \cdots \to \tau_n \to \mathtt{D}$ *for some* $\tau_1, \ldots, \tau_n$ *and*
3. *There exists an* i, $1 \leq i < m$ *such that* $E, \tau_i \vdash_{\mathrm{sp}} \phi(2pat_i) \Rightarrow T$ *can be derived using the inference rules in Figure 23.*

Intuitively, the sp predicate holds if (the first disjunct) we have a dynamic lambda, or (the second disjunct) we have a `fn` *match* where we cannot determine which branch to choose at specialization time — items 1 and 2 are merely used to get a hold on the patterns and the type we want (the type τ_n in the definition is the binding-time type of the patterns in the match rules), and item 3 formalizes what it means that we cannot determine which branch to choose at specialization time (recall that ϕ is the annotation-forgetting function, defined in Section 4.1.1). The idea is that if we have m match rules, then we cannot determine which branch to choose at specialization time if one of the first $m-1$ patterns contains a constant whose binding time is `D`. For example, if the function `f` in the declaration

```
val f  = fn {1 = 1, 2 = 2} => 1
          | {1 = x, 2 = y} => x
```

is applied to a pair with first component dynamic and second component static, i.e., binding-time type $\{1 \mapsto \mathtt{D}, 2 \mapsto \mathtt{S}\}$, we cannot determine which branch to choose at specialization time, and hence an sp-function should be inserted. This is reflected by the fact that there exists a deduction proving

$$E, \{1 \mapsto \mathtt{D}, 2 \mapsto \mathtt{S}\} \vdash_{\mathrm{sp}} \texttt{\{1=1,2=2\}} \Rightarrow T$$

[3] We use μ to range over the semantic object Type defined in [MTH90].

for arbitrary E.

In Figure 23 T is used for the truth value true, and F is used for false. The symbol ζ ranges over the truth values.

Two-Level Value Bindings $\boxed{E \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})}$

$$\frac{\begin{array}{c}\tau \neq \texttt{D} \qquad \neg\,\mathrm{sp}(\mathit{2exp}, E, \tau) \\ E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \texttt{C}) \qquad E \vdash \mathit{2exp} \Rightarrow \tau \qquad \langle E \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})\rangle\end{array}}{E \vdash \mathit{2pat}\ \texttt{=}\ \mathit{2exp}\ \langle\texttt{and}\ \mathit{2valbind}\rangle \Rightarrow (\mathit{VE}\langle +E'\rangle, \texttt{C}\langle \sqcup \mathit{rc}\rangle)} \tag{4.58}$$

$$\frac{\begin{array}{c}E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \texttt{D}, \texttt{R}) \\ \neg\,\mathrm{sp}(\mathit{2exp}, E, \tau) \qquad E \vdash \mathit{2exp} \Rightarrow (\texttt{D}, \_) \qquad \langle E \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})\rangle\end{array}}{E \vdash \mathit{2pat}\ \underline{\texttt{=}}\ \mathit{2exp}\ \langle\texttt{and}\ \mathit{2valbind}\rangle \Rightarrow (\mathit{VE}\langle +E'\rangle, \texttt{R})} \tag{4.59}$$

$$\frac{\begin{array}{c}\tau = \tau_1 \to \cdots \to \tau_n \to \texttt{D} \qquad \mathrm{sp}(\mathit{2exp}, E, \tau) \qquad n \geq 1 \\ \mathrm{TypeOf}(\mathit{2exp}) = \mu_1 \to \cdots \to \mu_n \to \mu \qquad \forall i : \text{equality-type}(\mu_i) \vee \tau_i = \texttt{D} \\ E \vdash \mathit{2pat} \Rightarrow (\mathit{VE}, \tau, \texttt{C}) \qquad E \vdash \mathit{2exp} \Rightarrow \tau \qquad \langle E \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})\rangle\end{array}}{E \vdash \mathit{2pat}\ \underline{\underline{\texttt{=}}}\ \mathit{2exp}\ \langle\texttt{and}\ \mathit{2valbind}\rangle \Rightarrow (\mathit{VE}\langle +E'\rangle, \texttt{R})} \tag{4.60}$$

$$\frac{E + E' \vdash \mathit{2valbind} \Rightarrow (E', \mathit{rc})}{E \vdash \texttt{rec}\ \mathit{2valbind} \Rightarrow (E', \mathit{rc})} \tag{4.61}$$

Figure 22: Inference Rules for Two-Level value bindings

Two-Level Data Type Bindings

The inference rules for two-level data type bindings are in Figure 24. The underlining expresses whether or not the constructors of the data type are known during specialization.

Two-Level Constructor Bindings

The inference rules for two-level constructor bindings are in Figure 25. The underlining expresses whether or not the constructors are known during specialization. Note that it is ensured that either all constructors of a data type are known or none are known.

Two-Level Exception Bindings

The inference rules for two-level exception bindings are in Figure 26. The underlining expresses whether or not the exception constructors are known during specialization. Note that we require that the argument of a specialization time exception constructor (if the exception constructor takes an argument) is either static or dynamic. The reason is that we want to be able to lift the argument, as explained in Section 3.9. See also Section 4.1.

$$\boxed{E, \tau \vdash_{\mathrm{sp}} atpat/patrow/pat \Rightarrow T/F}$$

$$\frac{}{E, \mathtt{S} \vdash_{\mathrm{sp}} scon \Rightarrow F} \qquad \frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} scon \Rightarrow T} \tag{4.62}$$

$$\frac{}{E, \tau \vdash_{\mathrm{sp}} var \Rightarrow F} \tag{4.63}$$

$$\frac{\tau \neq \mathtt{D}}{E, \tau \vdash_{\mathrm{sp}} con \Rightarrow F} \qquad \frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} con \Rightarrow T} \tag{4.64}$$

$$\frac{\tau \neq \mathtt{D}}{E, \tau \vdash_{\mathrm{sp}} excon \Rightarrow F} \qquad \frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} excon \Rightarrow T} \tag{4.65}$$

$$\frac{\tau \neq \mathtt{D} \qquad E, \tau \vdash_{\mathrm{sp}} patrow \Rightarrow \zeta}{E, \tau \vdash_{\mathrm{sp}} \mathtt{\{}patrow\mathtt{\}} \Rightarrow \zeta} \tag{4.66}$$

$$\frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} \mathtt{\{}\langle patrow\rangle\mathtt{\}} \Rightarrow T} \tag{4.67}$$

$$\frac{E, \tau \vdash_{\mathrm{sp}} \mathtt{(}\ pat\ \mathtt{)} \Rightarrow \zeta}{E, \tau \vdash_{\mathrm{sp}} atpat \Rightarrow \zeta} \tag{4.68}$$

$$\frac{\begin{array}{cc} \tau \neq \mathtt{D} & \varrho = \mathrm{BRecType\,of}\ \tau \\ E, \varrho(lab) \vdash_{\mathrm{sp}} pat \Rightarrow \zeta & \langle E, \tau \vdash_{\mathrm{sp}} patrow \Rightarrow \zeta'\rangle \end{array}}{E, \tau \vdash_{\mathrm{sp}} lab\ \mathtt{=}\ pat\ \langle\ \mathtt{,}\ patrow\rangle \Rightarrow \zeta\langle\vee\zeta'\rangle} \tag{4.69}$$

$$\frac{\tau \neq \mathtt{D} \qquad E(con) = (\tau', \_) \qquad E, \tau' \vdash_{\mathrm{sp}} atpat \Rightarrow \zeta}{E, \tau \vdash_{\mathrm{sp}} con\ atpat \Rightarrow \zeta} \tag{4.70}$$

$$\frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} con\ atpat \Rightarrow T} \tag{4.71}$$

$$\frac{\tau \neq \mathtt{D} \qquad E(excon) = (\tau', \_) \qquad E, \tau' \vdash_{\mathrm{sp}} atpat \Rightarrow \zeta}{E, \tau \vdash_{\mathrm{sp}} excon\ atpat \Rightarrow \zeta} \tag{4.72}$$

$$\frac{}{E, \mathtt{D} \vdash_{\mathrm{sp}} excon\ atpat \Rightarrow T} \tag{4.73}$$

$$\frac{}{E, \tau \vdash_{\mathrm{sp}} var\ \mathtt{as}\ pat \Rightarrow T} \tag{4.74}$$

Figure 23: Inference Rules used in the definition of the sp *predicate*

Two-Level Data Type Bindings $\boxed{E \vdash 2datbind \Rightarrow (TE, CE)}$

$$\frac{E \vdash 2conbind \Rightarrow (CE, \mathtt{C}) \qquad \langle E \vdash 2datbind \Rightarrow (TE', CE')\rangle}{E \vdash tycon \; \mathtt{=} \; 2conbind \; \langle \mathtt{and} \; 2datbind\rangle \Rightarrow (\{tycon \mapsto \mathtt{C}\}\langle + TE'\rangle, CE\langle + CE'\rangle)} \tag{4.75}$$

$$\frac{E \vdash 2conbind \Rightarrow (CE, \mathtt{R}) \qquad \langle E \vdash 2datbind \Rightarrow (TE', CE')\rangle}{E \vdash tycon \; \underline{\mathtt{=}} \; 2conbind \; \langle \mathtt{and} \; 2datbind\rangle \Rightarrow (\{tycon \mapsto \mathtt{R}\}\langle + TE'\rangle, CE\langle + CE'\rangle)} \tag{4.76}$$

Figure 24: Inference Rules for Two-Level data type bindings

Two-Level Constructor Bindings $\boxed{E \vdash 2conbind \Rightarrow (CE, rc)}$

$$\frac{\langle E \vdash 2ty \Rightarrow \tau\rangle \qquad \langle\langle E \vdash 2conbind \Rightarrow (CE, rc)\rangle\rangle}{E \vdash con \; \langle \mathtt{of} \; 2ty\rangle \; \langle \; \mathtt{|} \; 2conbind\rangle \Rightarrow (\{con \mapsto \mathtt{C}\}\langle + \{con \mapsto (\tau, \mathtt{C})\}\rangle\langle\langle + CE\rangle\rangle, \mathtt{C}\langle\langle \sqcup rc\rangle\rangle)} \tag{4.77}$$

$$\frac{\langle E \vdash 2ty \Rightarrow \mathtt{D}\rangle \qquad \langle\langle E \vdash 2conbind \Rightarrow (CE, rc)\rangle\rangle}{E \vdash con \; \langle \underline{\mathtt{of}} \; 2ty\rangle \; \langle\langle \; \underline{\mathtt{|}} \; 2conbind\rangle\rangle \Rightarrow (\{con \mapsto \mathtt{R}\}\langle + \{con \mapsto (\mathtt{D}, \mathtt{R})\}\rangle\langle\langle + CE\rangle\rangle, \mathtt{R})} \tag{4.78}$$

Figure 25: Inference Rules for Two-Level constructor bindings

Two-Level Exception Bindings $\boxed{E \vdash 2exbind \Rightarrow (EE, rc)}$

$$\frac{\langle E \vdash 2ty \Rightarrow \tau \qquad \tau \in \{\mathtt{S}, \mathtt{D}\}\rangle \qquad \langle\langle E \vdash 2exbind \Rightarrow (EE, rc)\rangle\rangle}{E \vdash excon \; \langle \mathtt{of} \; 2ty\rangle \; \langle\langle \mathtt{and} \; 2exbind\rangle\rangle \Rightarrow (\{excon \mapsto \mathtt{C}\}\langle + \{excon \mapsto (\tau, \mathtt{C})\}\rangle\langle\langle + EE\rangle\rangle, \mathtt{C}\langle\langle \sqcup rc\rangle\rangle)} \tag{4.79}$$

$$\frac{\langle E \vdash 2ty \Rightarrow \mathtt{D}\rangle \qquad \langle\langle E \vdash 2exbind \Rightarrow (EE, rc)\rangle\rangle}{E \vdash excon \; \langle \underline{\mathtt{of}} \; 2ty\rangle \; \langle\langle \mathtt{and} \; 2exbind\rangle\rangle \Rightarrow (\{excon \mapsto \mathtt{R}\}\langle + \{excon \mapsto (\mathtt{D}, \mathtt{R})\}\rangle\langle\langle + EE\rangle\rangle, \mathtt{R})} \tag{4.80}$$

Figure 26: Inference Rules for Two-Level exception bindings

Two-Level Type Expressions

The inference rules for two-level type expressions are in Figure 27.

The annotations in type expressions tells which parts of an argument of a constructor are known during specialization. Two-level type expression only occur in constructor bindings in the grammar rules (with the exception of type expression rows). In rules 4.83–4.84 we see the reason why type constructors are bound in the type environment; if the constructors of a data type are known during elaboration (rule 4.83) the values of that data type will have binding time `C`. If the constructors are not known, the values of the data type will have binding time `D`, hence the definition of 4.84.

Two-Level Types $\boxed{E \vdash \mathit{2ty} \Rightarrow \tau}$

$$\frac{\langle E \vdash \mathit{2tyrow} \Rightarrow (\varrho, \mathtt{C})\rangle}{E \vdash \mathtt{\{}\ \langle \mathit{2tyrow}\rangle\ \mathtt{\}} \Rightarrow \{\}\langle +\varrho\rangle} \tag{4.81}$$

$$\frac{\langle E \vdash \mathit{2tyrow} \Rightarrow (\mathtt{D}, \mathtt{R})\rangle}{E \vdash \underline{\mathtt{\{}}\ \langle \mathit{2tyrow}\rangle\ \underline{\mathtt{\}}} \Rightarrow \mathtt{D}} \tag{4.82}$$

$$\frac{E(\mathit{tycon}) = \mathtt{C}}{E \vdash \mathit{tycon} \Rightarrow \mathtt{C}} \tag{4.83}$$

$$\frac{E(\mathit{tycon}) = \mathtt{R}}{E \vdash \underline{\mathit{tycon}} \Rightarrow \mathtt{D}} \tag{4.84}$$

$$\frac{E \vdash \mathit{2ty} \Rightarrow \tau \qquad \tau \neq \mathtt{D}}{E \vdash \mathtt{(}\ \mathit{2ty}\ \mathtt{)} \Rightarrow \tau} \tag{4.85}$$

$$\frac{E \vdash \mathit{2ty} \Rightarrow \mathtt{D}}{E \vdash \underline{\mathtt{(}}\ \mathit{2ty}\ \underline{\mathtt{)}} \Rightarrow \mathtt{D}} \tag{4.86}$$

$$\frac{E \vdash \mathit{2ty}_1 \Rightarrow \tau_1 \qquad E \vdash \mathit{2ty}_2 \Rightarrow \tau_2}{E \vdash \mathit{2ty}_1\ \mathtt{->}\ \mathit{2ty}_2 \Rightarrow \tau_1 \rightarrow \tau_2} \tag{4.87}$$

$$\frac{E \vdash \mathit{2ty}_1 \Rightarrow \mathtt{D} \qquad E \vdash \mathit{2ty}_2 \Rightarrow \mathtt{D}}{E \vdash \mathit{2ty}_1\ \underline{\mathtt{->}}\ \mathit{2ty}_2 \Rightarrow \mathtt{D}} \tag{4.88}$$

Figure 27: Inference Rules for Two-Level type expressions

Two-Level Type Expression Rows

The inference rules for two-level type expression rows are in Figure 28. The only thing to notice is that if the record type, c.f. rule 4.82, is dynamic then all its components are too.

Two-Level Type Rows $\boxed{E \vdash \mathit{2tyrow} \Rightarrow (\tau, rc)}$

$$\frac{E \vdash \mathit{2ty} \Rightarrow \tau \qquad \langle E \vdash \mathit{2tyrow} \Rightarrow (\varrho, \mathtt{C})\rangle}{E \vdash \mathit{lab}\ :\ \mathit{2ty}\ \langle\ ,\ \mathit{2tyrow}\rangle \Rightarrow (\{\mathit{lab} \mapsto \tau\}\langle+\varrho\rangle, \mathtt{C})} \tag{4.89}$$

$$\frac{E \vdash \mathit{2ty} \Rightarrow \mathtt{D} \qquad \langle E \vdash \mathit{2tyrow} \Rightarrow (\varrho, \mathtt{R})\rangle}{E \vdash \mathit{lab}\ \underline{:}\ \mathit{2ty}\ \langle\ \underline{,}\ \mathit{2tyrow}\rangle \Rightarrow (\{\mathit{lab} \mapsto \mathtt{D}\}\langle+\varrho\rangle, \mathtt{R})} \tag{4.90}$$

Figure 28: Inference Rules for Two-Level type-expression rows

Two-Level Programs

Finally we turn to two-level programs which are just top-level declarations. The inference rule is seen in Figure 29. This is quite an abuse of notation as programs *are* declarations,

Two-Level Programs $\boxed{E \vdash \mathit{2dec}_{tl} \Rightarrow (E', rc)}$

$$\frac{E \vdash \mathit{2dec} \Rightarrow (E', rc) \qquad \mathrm{exnuni}(\mathit{EE} \text{ of } E')}{E \vdash \mathit{2dec}_{tl} \Rightarrow (E', rc)} \tag{4.91}$$

Figure 29: Inference rule for Two-Level Programs

but we ask the reader to bear with us.

The semantic operation "exnuni" is defined below and is used to express that either all exception constructors are are known at compile-time or they are all deferred to runtime. (See Section 3.9 for an explanation of why this restriction is made). We must enforce the restriction at the declaration level as the *excon* type is defined little by little. Because all exceptions are defined at top-level there are no local ones to require uniformity of.

Definition 6 (The exnuni **predicate)** *The exception uniformity predicate* $\mathrm{exnuni}(\mathit{EE})$ *is satisfied if and if only*

$$\forall \mathit{excon} \in \mathrm{Dom}(\mathit{EE}) : \exists \tau : \mathit{EE}(\mathit{excon}) = (\tau, \mathtt{C}) \vee \mathit{EE}(\mathit{excon}) = \mathtt{C}$$

or $\forall \mathit{excon} \in \mathrm{Dom}(\mathit{EE}) : \mathit{EE}(\mathit{excon}) = \mathtt{R}$.

4.3 Dynamic Semantics for the Two-Level Simplified Bare Language

In this section we define the dynamic semantics for the Two-Level Simplified Bare Language. This specifies the compiler generator.

The general framework is the following. We assume given a well-annotated two-level program *2dec*, and define a bunch of semantic functions and operations; the section ends with defining the function $\mathcal{C}_{program}$ which gives the semantics of a two-level program. Besides the semantic operations we shall assume the existence of some SML functions which the generating extension can use — these functions will be described here and are declared in an "include file" the contents of which is shown in Appendix A. Moreover, we shall assume the existence of an SML data type which can be used to represent code in the generating extension (like the one in Section 3.3).

Types for Two-Level Semantic Functions

We define one semantic function $\mathcal{C}_{phrase}$ for each syntactic two-level phrase class. Most of these functions (except those for declarations and value, data type, and exception bindings) will be typed in a uniform manner: they will take a *2phrase* and return either a one-level *phrase* or an expression (a piece of code) which *evaluates to* a *phrase*. We use $\underline{phrase}$ to denote the type *exp* of expressions (code) which evaluates to *phrase* [Jon91] [JGS93]. This choice has been made such that it is simple to ensure type-correctness of the generating extension. We conjecture (strongly), but have not proved, that the generating extension and the residual program are always well-typed according to the static semantics of SML.

The choice of notation implies that the type of for instance $\mathcal{C}_{2atexp}$ can be written as

$$\mathcal{C}_{2atexp} : 2atexp \rightarrow atexp/\underline{atexp}$$

meaning that it takes a *2atexp* as argument as gives either an *atexp* or an $\underline{atexp}$ as result. In other words the result is an atomic expression either at specialization time or at residual time.

Notation

Our meta-notation will resemble Standard ML. Moreover, we will use the notation for semantic objects and operations employed in the Definition of Standard ML [MTH90].

Moreover, we shall generally omit injections and projections in the meta-language to avoid cluttering the notation. We conjecture, but have not proved, that this is always safe, or in other words that: for any well-annotated two-level program *2dec* for which it holds that $dec = \phi(2dec)$ is well-typed according to SML static semantics, the dynamic semantics does not "go wrong." We return to this correctness issue in Section 4.4.

Recall from earlier that we use the `fixed-spacing` font for ML-code or more precisely their abstract syntax trees. Such syntax trees are constants and thus evaluate to themselves. We use the notation $\lceil$`ml-code`$\rceil$ for the syntax tree of an expression that evaluates to the syntax tree of `ml-code`. Some components inside a $\lceil\ \rceil$ may be dot-underlined meaning that they must be evaluated (at cogen-time) instead and their values inserted instead. No evaluating expressions are generated for such values so dot-underlined ML-code goes into the generating extension whereas ML-code inside $\lceil\ \rceil$ normally goes into the residual program.

Two-Level Atomic Expressions

The semantics for two-level atomic expressions is given in Figure 31.

Note how the static atomic expressions are essentially copied to the generating extension. Here we see the reason for the two different kinds of variables to be bound at specialization time. Consider the case for $\underline{var}$, which to recall is used for a variable var that will be bound to something dynamic (i.e., something of type exp) at specialization time. If $\mathcal{C}_{2atexp}[\![\underline{var}]\!]$ were simply defined to be var the result type would be $\underline{exp}$; by inserting a parentheses at specialization time around the code the variable will be bound to, the result type becomes $\underline{atexp}$ as we wish. As the backquote notation is perhaps a bit heavy, we show the syntax tree for $\lceil(\underset{\cdots}{var})\rceil$ in Figure 30. Note that `PAR1atexp` is inside

The notation $\lceil(\underset{\cdots}{var})\rceil$ is shorthand for:

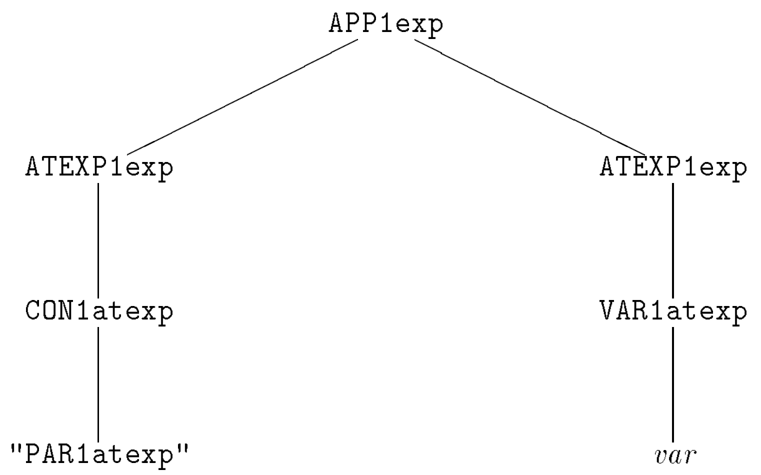

Figure 30: Example of backquote notation.

quotation marks, while var is not. This means that when the compiler generator is type checked the ML system only checks the types of the specialization time syntax tree; the residual part of the syntax tree (as the `PAR1atexp`) tag can use misspelled constructors et cetera without the warning from the type checker. Such an error will first show itself when the misspelled constructor appears in a generating extension and that extension is type checked.

The most complicated case for $\mathcal{C}_{2atexp}$ is the one for a residual `let`-expression. It has been explained in Section 3.4.4. We just remark that $\lceil\ \rceil$ is used for "a piece of code which evaluates to the empty declaration," that is, the syntax tree `EMPTY1dec`. Also notice that

$\mathcal{C}_{2dec}[\![2dec]\!]$ returns a declaration declaring the `SeenB4list`, `code`, `getcode` and `sp-gen` function shown in Section 3.4.4.

$$\mathcal{C}_{2atexp} : 2atexp \rightarrow atexp/\underline{atexp}$$

$$\begin{array}{lcl}
\mathcal{C}_{2atexp}[\![scon]\!] & = & scon \\
\mathcal{C}_{2atexp}[\![var]\!] & = & var \\
\mathcal{C}_{2atexp}[\![\underline{var}]\!] & = & \lceil(\underline{var})\rceil \\
\mathcal{C}_{2atexp}[\![\underline{\underline{var}}]\!] & = & \lceil var\rceil \\
\mathcal{C}_{2atexp}[\![con]\!] & = & con \\
\mathcal{C}_{2atexp}[\![\underline{con}]\!] & = & \lceil con\rceil \\
\mathcal{C}_{2atexp}[\![excon]\!] & = & excon \\
\mathcal{C}_{2atexp}[\![\underline{excon}]\!] & = & \lceil excon\rceil \\
\mathcal{C}_{2atexp}[\![\{\langle 2exprow\rangle\}]\!] & = & \{\langle\mathcal{C}_{2exprow}[\![2exprow]\!]\rangle\} \\
\mathcal{C}_{2atexp}[\![\underline{\{}\langle 2exprow\rangle\underline{\}}]\!] & = & \lceil\{\langle\underline{\mathcal{C}_{2exprow}[\![2exprow]\!]}\rangle\}\rceil \\
\mathcal{C}_{2atexp}[\![\mathtt{let}\ 2dec\ \mathtt{in}\ 2exp\ \mathtt{end}]\!] & = & \mathtt{let}\ \mathcal{C}_{2dec}[\![2dec]\!]\ \mathtt{in}\ \mathcal{C}_{2exp}[\![2exp]\!]\ \mathtt{end} \\
\mathcal{C}_{2atexp}[\![\underline{\mathtt{let}}\ 2dec\ \underline{\mathtt{in}}\ 2exp\ \underline{\mathtt{end}}]\!] & = &
\end{array}$$

$$\begin{array}{l}
\quad \mathtt{let\ val\ getcode\ =\ fn\ ()\ =>}\ \lceil\ \rceil \\
\qquad \mathcal{C}_{2dec}[\![2dec]\!] \\
\qquad \mathtt{val\ body\ =}\ \mathcal{C}_{2exp}[\![2exp]\!] \\
\quad \mathtt{in}\ \ \lceil\mathtt{let}\ \underline{\mathtt{getcode()}}\ \mathtt{in}\ \underline{\mathtt{body}}\ \mathtt{end}\rceil \\
\quad \mathtt{end}
\end{array}$$

$$\begin{array}{lcl}
\mathcal{C}_{2atexp}[\![(\ 2exp\ )]\!] & = & (\mathcal{C}_{2exp}[\![2exp]\!]) \\
\mathcal{C}_{2atexp}[\![\underline{(}\ 2exp\ \underline{)}]\!] & = & \lceil(\underline{\mathcal{C}_{2exp}[\![2exp]\!]})\rceil
\end{array}$$

Figure 31: Semantics for two-level atomic expressions

Two-Level Expression Rows

The semantics for two-level expression rows is given in Figure 32; it is straightforward.

$$\mathcal{C}_{2exprow} : 2exprow \rightarrow exprow/\underline{exprow}$$

$$\begin{array}{lcl}
\mathcal{C}_{2exprow}[\![lab\ \mathtt{=}\ 2exp\ \langle\ ,\ 2exprow\rangle]\!] & = & lab\ \mathtt{=}\ \mathcal{C}_{2exp}[\![2exp]\!]\ \langle\ ,\ \mathcal{C}_{2exprow}[\![2exprow]\!]\rangle \\
\mathcal{C}_{2exprow}[\![lab\ \underline{\mathtt{=}}\ 2exp\ \langle\ \underline{,}\ 2exprow\rangle]\!] & = & \lceil lab\ \mathtt{=}\ \underline{\mathcal{C}_{2exp}[\![2exp]\!]}\ \langle\ ,\ \underline{\mathcal{C}_{2exprow}[\![2exprow]\!]}\rangle\rceil
\end{array}$$

Figure 32: Semantics for two-level expression rows

Two-Level Expressions

The semantics for two-level expressions is given in Figure 33.

Notice how backquotes are used in the case for the underlined atomic expression; intuitively the idea is (as was the case with underlined variable, see above) to make sure that the types are correct. This ensures that an atomic expression in the generating extension is converted into an expression.

The cases for applications are straightforward.

For static lambda abstractions we use the operator $\mathcal{C}'_{2match}$ which is used to make a lambda abstraction which stages the decomposition of values as described in Section 3.6.2. The operator is defined in Figure 34 and uses the operators defined in Figures 35–38.

The operation $\mathcal{C}'_{2match}$ is used to generate the code we wish for decomposition of values using pattern matching, c.f. Section 3.6.2. It is defined in Figure 34.

For `handle` and <u>`handle`</u> we generate the same code. This makes sense as we always generate code to handle the exception both at specialzation time and at residual time. Note by the way the use of a meta-let (cogen-time `let`-expression). The status of the let-expression is determined by the font its written in. The function `mk_exn` used in the rules is defined on page 105. The rules for `raise` expressions are straightforward.

In the case of an application of a lift operator we use the semantic operation LiftExp which is defined as follows.

Definition 7 (The LiftExp **operator)** *The operator* LiftExp : Type $\rightarrow$ *exp* $\rightarrow$ <u>*exp*</u> *is a partial function defined by*

$$\mathrm{LiftExp}\,\tau\,exp = \begin{cases} \texttt{lift\_int}\,exp, & \mathit{if}\ \tau = \texttt{int} \\ \texttt{lift\_real}\,exp, & \mathit{if}\ \tau = \texttt{real} \\ \texttt{lift\_string}\,exp, & \mathit{if}\ \tau = \texttt{string} \end{cases}$$

where `lift_int`*,* `lift_int`*, and* `lift_int` *are Standard ML functions defined in the include file (Appendix A), which given a basic value b return a piece of code of type* <u>*exp*</u> *which, when evaluated according to the dynamic semantics of SML, yields the basic value b.*

Intuitively, LiftExp simply chooses the correct lift function to be applied in the generating extension on the basis of the SML type of the expression that shall be lifted.

Two-Level Matches

The semantics for two-level matches is given in Figure 39; it is straightforward.

Two-Level Match Rules

The semantics for two-level match rules is given in Figure 40; it is straightforward.

Two-Level Declarations

The semantics for two-level declarations is given in Figure 41.

$$\mathcal{C}_{2exp} : \mathit{2exp} \rightarrow \mathit{exp}/\underline{\mathit{exp}}$$

$$
\begin{array}{lcl}
\mathcal{C}_{2exp}[\![\texttt{atexpexp}\ \mathit{2atexp}]\!] & = & \mathcal{C}_{2atexp}[\![\mathit{2atexp}]\!] \\
\mathcal{C}_{2exp}[\![\underline{\texttt{atexpexp}}\ \mathit{2atexp}]\!] & = & \lceil \mathcal{C}_{2atexp}[\![\mathit{2atexp}]\!] \rceil \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \texttt{@}\ \mathit{2exp}_2]\!] & = & \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \\
\mathcal{C}_{2exp}[\![\mathit{2exp}_1\ \underline{\texttt{@}}\ \mathit{2exp}_2]\!] & = & \lceil \mathcal{C}_{2exp}[\![\mathit{2exp}_1]\!]\ \mathcal{C}_{2exp}[\![\mathit{2exp}_2]\!] \rceil \\
\mathcal{C}_{2exp}[\![\mathit{2exp}\ \texttt{handle}\ \mathit{2match}]\!] & = & \\
\quad \text{let } e = \mathcal{C}_{2exp}[\![\mathit{2exp}]\!]\ \texttt{handle x => mk\_exn x} & & \\
\quad \text{in } \lceil e\ \texttt{handle}\ \mathcal{C}_{2match}[\![\mathit{2match}]\!] \rceil & & \\
\mathcal{C}_{2exp}[\![\mathit{2exp}\ \underline{\texttt{handle}}\ \mathit{2match}]\!] & = & \mathcal{C}_{2exp}[\![\mathit{2exp}\ \texttt{handle}\ \mathit{2match}]\!] \\
\mathcal{C}_{2exp}[\![\texttt{raise}\ \mathit{2exp}]\!] & = & \texttt{raise}\ \mathcal{C}_{2exp}[\![\mathit{2match}]\!] \\
\mathcal{C}_{2exp}[\![\underline{\texttt{raise}}\ \mathit{2exp}]\!] & = & \lceil \texttt{raise}\ \mathcal{C}_{2exp}[\![\mathit{2match}]\!] \rceil \\
\mathcal{C}_{2exp}[\![\texttt{fn}\ \mathit{2match}]\!] & = & \texttt{fn}\ \mathcal{C}'_{2match}[\![\mathit{2match}]\!] \\
\mathcal{C}_{2exp}[\![\underline{\texttt{fn}}\ \mathit{2match}]\!] & = & \lceil \texttt{fn}\ \mathcal{C}_{2match}[\![\mathit{2match}]\!] \rceil \\
\mathcal{C}_{2exp}[\![\texttt{lift\_exp}\ \mathit{2exp}]\!] & = & \mathrm{LiftExp}\ (\mathrm{TypeOf}\ \ \mathit{2exp})\ \mathcal{C}_{2exp}[\![\mathit{2exp}]\!]
\end{array}
$$

Figure 33: Semantics for two-level expressions

$$\mathcal{C}'_{2match} : \mathit{2match} \rightarrow \mathit{match}$$

$$\mathcal{C}'_{2match}[\![\mathit{2mrule}\ \langle\ \texttt{|}\ \mathit{2match}\rangle]\!] \quad = \quad \mathcal{C}'_{2mrule}[\![\mathit{2mrule}]\!]\ \langle\ \texttt{|}\ \mathcal{C}'_{2match}[\![\mathit{2match}]\!]\rangle$$

Figure 34: The $\mathcal{C}'_{2match}$ operator

$$\mathcal{C}'_{2mrule} : \mathit{2mrule} \rightarrow \mathit{mrule}$$

$$
\begin{array}{l}
\mathcal{C}'_{2mrule}[\![\mathit{2pat}\ \texttt{=>}\ \mathit{2exp}]\!] \quad = \\
\quad \text{let } (\mathit{pat},[\mathit{pat}_1,\ldots,\mathit{pat}_n],[\mathit{exp}_1,\ldots,\mathit{exp}_n]) = \mathcal{C}'_{2pat}[\![\mathit{2pat}]\!]\ [\,]\ [\,] \text{ in} \\
\qquad \text{if } n = 0 \text{ then } \mathcal{C}_{2mrule}[\![\mathit{2pat}\ \texttt{=>}\ \mathit{2exp}]\!] \\
\qquad \text{else } \mathit{pat}\ \texttt{=>}\ \lceil \mathcal{C}_{2exp}[\![\underline{\texttt{fn}}\ \{\texttt{1 = }\mathit{pat}_1\texttt{, ... , n = }\mathit{pat}_n\}\ \texttt{=>}\ \mathit{2exp}]\!] \\
\qquad\qquad\qquad \{\texttt{1 = }\mathit{exp}_1\texttt{, ... , n = }\mathit{exp}_n\} \rceil
\end{array}
$$

Figure 35: The $\mathcal{C}'_{2mrule}$ operator

$\mathcal{C}'_{2pat} : \mathit{2pat} \rightarrow \mathit{2pat\ list} \rightarrow \mathit{exp\ list} \rightarrow \mathit{pat} \times \mathit{2pat\ list} \times \mathit{exp\ list}$

$\mathcal{C}'_{2pat}[\![\mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el} \quad = \quad \mathcal{C}'_{2atpat}[\![\mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el}$

$\mathcal{C}'_{2pat}[\![\mathit{con}\ \mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el} \quad =$

$\quad \text{let } (\mathit{pat},\mathit{pl}',\mathit{el}') = \mathcal{C}'_{2atpat}[\![\mathit{2pat}]\!]\ []\ []\ \text{in } (\mathit{con}\ \mathit{pat},\ \mathit{pl}',\ \mathit{el}')$

$\mathcal{C}'_{2pat}[\![\underline{\mathit{con}}\ \mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el} \quad =$

$\quad \text{let } \mathit{var} \text{ be a fresh variable in } (\mathit{var},[\mathit{2atpat}]\ @\ \mathit{pl},\ [\mathit{var}]\ @\ \mathit{el})$

$\mathcal{C}'_{2pat}[\![\mathit{excon}\ \mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el} \quad = \quad (\text{* as value constructors *})$

$\mathcal{C}'_{2pat}[\![\underline{\mathit{excon}}\ \mathit{2atpat}]\!]\ \mathit{pl}\ \mathit{el} \quad = \quad (\text{* as value constructors *})$

Figure 36: The $\mathcal{C}'_{2pat}$ operator

$\mathcal{C}'_{2atpat} : \mathit{2atpat} \rightarrow \mathit{2pat\ list} \rightarrow \mathit{exp\ list} \rightarrow \mathit{atpat} \times \mathit{2pat\ list} \times \mathit{exp\ list}$

$\mathcal{C}'_{2atpat}[\![\{\langle \mathit{2patrow}\rangle\}]\!]\ \mathit{pl}\ \mathit{el} \quad =$

$\quad \text{let } \langle(\mathit{patrow},\mathit{pl}',\mathit{el}') = \mathcal{C}'_{2patrow}[\![\mathit{2patrow}]\!]\ []\ []\rangle \text{ in } (\{\},\mathit{pl},\mathit{el})\ \langle +(\{\mathit{patrow}\},\mathit{pl}',\mathit{el}')\rangle$

$\mathcal{C}'_{2atpat}[\![\underline{\{}\langle \mathit{2patrow}\rangle\underline{\}}]\!]\ \mathit{pl}\ \mathit{el} \quad =$

$\quad \text{let } \mathit{var} \text{ be a fresh variable in } (\mathit{var},[\{\langle \mathit{2patrow}\rangle\}]\ @\ \mathit{pl},\ [\mathit{var}]\ @\ \mathit{el})$

$\mathcal{C}'_{2atpat}[\![(\ \mathit{2pat}\ )]\!]\ \mathit{pl}\ \mathit{el} \quad =$

$\quad \text{let } (\mathit{pat},\mathit{pl}',\mathit{el}') = \mathcal{C}'_{2pat}[\![\mathit{2pat}]\!]\ \mathit{pl}\ \mathit{el} \text{ in } ((\ \mathit{pat}\ ),\mathit{pl}',\mathit{el}')$

$\mathcal{C}'_{2atpat}[\![\mathrm{x}]\!]\ \mathit{pl}\ \mathit{el} \quad = \quad (\mathrm{x},\mathit{pl},\mathit{el})$

Figure 37: The $\mathcal{C}'_{2atpat}$ operator

$\mathcal{C}'_{2patrow} : \mathit{2patrow} \rightarrow \mathit{2pat\ list} \rightarrow \mathit{exp\ list} \rightarrow \mathit{patrow} \times \mathit{2pat\ list} \times \mathit{exp\ list}$

$\mathcal{C}'_{2patrow}[\![\mathit{lab} = \mathit{2pat}\ \langle\ ,\ \mathit{2patrow}\rangle]\!] \quad =$

$\quad \text{let } (\mathrm{pat},\mathit{pl}',\mathit{el}') = \mathcal{C}'_{2pat}[\![\mathit{2pat}]\!]\ \mathit{pl}\ \mathit{el}$

$\qquad \langle(\mathit{patrow},\mathit{pl}'',\mathit{el}'') = \mathcal{C}'_{2patrow}[\![\mathit{2patrow}]\!]\ \mathit{pl}\ \mathit{el}\rangle$

$\quad \text{in } (\mathit{lab} = \mathit{pat}\ \langle\ ,\ \mathit{patrow}\ \rangle,\ \mathit{pl}'\ \langle +\mathit{pl}''\rangle,\ \mathit{el}'\ \langle +\mathit{el}''\rangle)$

Figure 38: The $\mathcal{C}'_{2patrow}$ operator

$$\mathcal{C}_{2match} : 2match \rightarrow match/\underline{match}$$

$$\begin{array}{lcl}
\mathcal{C}_{2match}[\![2mrule \ \langle \ \texttt{|} \ 2match \rangle]\!] & = & \mathcal{C}_{2mrule}[\![2mrule]\!] \ \langle \ \texttt{|} \ \mathcal{C}_{2match}[\![2match]\!] \rangle \\
\mathcal{C}_{2match}[\![2mrule \ \langle \ \underline{\texttt{|}} \ 2match \rangle]\!] & = & \lceil \mathcal{C}_{2mrule}[\![2mrule]\!] \ \langle \ \texttt{|} \ \mathcal{C}_{2match}[\![2match]\!] \rangle \rceil
\end{array}$$

Figure 39: Semantics for two-level matches

$$\mathcal{C}_{2mrule} : 2mrule \rightarrow mrule/\underline{mrule}$$

$$\begin{array}{lcl}
\mathcal{C}_{2mrule}[\![2pat \ \texttt{=>} \ 2exp]\!] & = & \mathcal{C}_{2pat}[\![2pat]\!] \ \texttt{=>} \ \mathcal{C}_{2exp}[\![2exp]\!] \\
\mathcal{C}_{2mrule}[\![2pat \ \underline{\texttt{=>}} \ 2exp]\!] & = & \lceil \mathcal{C}_{2pat}[\![2pat]\!] \ \texttt{=>} \ \mathcal{C}_{2exp}[\![2exp]\!] \rceil
\end{array}$$

Figure 40: Semantics for two-level match rules

Consider first the case for value declarations. The call to $\mathcal{C}_{2valbind}$ yields a value binding and a so-called *seenlist*, which is a list of names for sp-functions and corresponding SML types for the elements in the SeenB4List for the sp-function. The *seenlist* semantic object (we also use *seenlist* to range over this semantic object for simplicity) is defined as

$$seenlist = (var \times \text{Type})\ list$$

where Type is the semantic object for SML types defined in [MTH90]. The semantic operation mk_seenB4list is as follows (two semantic operations are used in the definition; they are defined below).

Definition 8 (The mk_seenB4list **operator)** *The code accumulator generating operator,* mk_seenB4list : *seenlist* $\rightarrow$ *valbind, is defined as follows.*

case *seenlist* of
  [] =>
    `val code = new_code()`
    `val getcode = fn () => SEQ1dec(getcode(),!code)`
 | $((var_{sp}, \tau) :: seenlist)$ =>
    `val` (mk_seenB4list_name var_{sp}) `= ref ([]:` (Type2TypeExpression τ) `list);`
    mk_seenB4list *seenlist*

The `SEQ1dec` used in the definition is a constructor of the data type used for declarations in the generation extension; see Section 3.3. The only reason the seenlist contains SML types is because of the way references are typed in SML — we must be able to ascribe the empty list a type (see [Tof90] for an excellent description of typing rules for references used in SML). So this operation sets up a code list for the specialization functions in the value binding and one SeenB4List for each sp-function.

Definition 9 (The mk_seenB4list_name **operator)** *The naming operator for accumulators of static arguments,* mk_seenB4list_name : *var* $\rightarrow$ *exp, takes a variable var as argument and returns an expression consisting of a variable var*`_seenB4list`*.*

Definition 10 (The Type2TypeExpression **operator)** *Given a two-level program 2dec let dec* $= \phi(2dec)$. *Let E be the environment obtained by elaboration of dec according to the static semantics of SML. By the syntactic restrictions on data type declarations, there is a one-to-one correspondence between type names in the type environment of E and the set of type constructors in the dec. Due to this correspondence, it is now straightforward to define the operator* Type2TypeExpression : Type $\rightarrow$ *ty which for a given monomorphic SML type* τ *inferred during elaboration of the dec yields a type expression ty which, according to the static semantics of SML [MTH90, Section 4], elaborates to* τ. *We omit the details.*

Now consider the case for data type declarations. The first component returned from the call to $\mathcal{C}_{2datbind}$ contains the data types with known constructors, the other component the data types with unknown constructors. The code generated ensures that the data types with known constructors are declared in the generating extension and that the data types with unknown constructors are declared in the residual program. The SML function `add_to_code` is defined in the include file and merely appends a piece of code to the codelist `code`.

For exception declarations we generate code to declare the exception both in the generating extension and in the residual program.

Sequential and empty declarations are straightforward.

Two-Level Value Bindings

The semantics for two-level value bindings is given in Figures 42 and 43.

The first cases for specialization time and residual value bindings are easy. For specialization functions we have already described the ideas in Chapter 3. The semantic operation BTypeOf is defined as follows.

Definition 11 (The BTypeOf **operator)***Assume given a two-level program 2dec. Then for any 2exp or 2match phrase in 2dec the operator* BTypeOf : *2exp* $\cup$ *2match* $\rightarrow$ BType *yields the binding-time type of the phrase given by the elaboration of the 2dec in the initial static two-level basis as defined by the static semantics in Section 4.2.*

It other words it returns the binding-time type (τ), which is used to define `sp_do` functions.

The function `mk_exn` used in Figure 43 is defined on page 105.

The semantic operation mk_sp_do_fun is defined in Figure 44 (where we again use μ to range over the semantic object Type defined in [MTH90]). It uses the ideas from Chapter 3. In the definition we use two "global variables," σ and υ. We will allow ourselves to update these destructively for notational convenience. The variable σ is used for a set of tuples $((\mu, \tau), var_{do})$ relating an SML type and binding-time type combination to an `sp_do` function name. It is simply used as a compiler generation time "seen before map." The variable υ is used for a value binding; it will contain the generated `sp_do` functions. Note that we use a simple numbering scheme to encode positions, c.f. Chapter 3, where a sequence of numbers separated by underscores encodes the position.

The semantic operation split_pat is defined in Figure 48 and uses the definitions of split_atpat and split_patrow defined in Figures 46 and 47, and the `makestring` function which simply turns a number into the corresponding digit-string. The operation takes

$\mathcal{C}_{2dec} : \mathit{2dec} \rightarrow \mathit{dec}$

$\mathcal{C}_{2dec}[\![\texttt{val}\ \mathit{2valbind}]\!] =$
case $\mathcal{C}_{2valbind}[\![\mathit{2valbind}]\!]$ of
([], *valbind*) =>
`val` valbind
| (*seenlist*, *valbind*) =>
mk_seenB4list *seenlist* ;
`val` *valbind*

$\mathcal{C}_{2dec}[\![\texttt{datatype}\ \mathit{2datbind}]\!] =$
case $\mathcal{C}_{2datbind}[\![\mathit{2datbind}]\!]$ of
(Some *datbind_opt*,None) =>
`datatype` *datbind_opt*
| (None,Some *datbind_opt'*) =>
`val _ = add_to_code code` ⌈`datatype(`*datbind_opt'*`)`⌉
| (Some *datbind_opt*,Some *datbind_opt'*) =>
`datatype` *datbind_opt* ;
`val _ = add_to_code code` ⌈`datatype` (*datbind_opt'*)⌉

$\mathcal{C}_{2dec}[\![\texttt{exception}\ \mathit{2exbind}]\!] =$
let (c,r) = $\mathcal{C}_{2exbind}[\![\mathit{2exbind}]\!]$ in
`val _ = add_to_code code` ⌈`exception` (r)⌉ ;
`exception` c

$\mathcal{C}_{2dec}[\![\quad]\!] =$

$\mathcal{C}_{2dec}[\![\mathit{2dec}_1\ \langle;\rangle\ \mathit{2dec}_2]\!] = \mathcal{C}_{2dec}[\![\mathit{2dec}_1]\!]\ ;\ \mathcal{C}_{2dec}[\![\mathit{2dec}_2]\!]$

Figure 41: Semantics for two-level declarations

a pattern and returns a the dynamic part of the pattern and a list of the static parts together with their positions. The semantic operation is used to provide arguments to `mk_sp_res_pat` and to generate the residual pattern, as seen in Figure 42 and as explained in Section 3.6.2.

Two-Level Atomic Patterns

The semantics for two-level atomic patterns is given in Figure 49. The cases are all straightforward.

Two-Level Pattern rows

The semantics for two-level pattern rows is given in Figure 50. The cases are all straightforward.

Two-Level Patterns

The semantics for two-level patterns is given in Figure 51. The cases are all straightforward.

Two-Level Data Type Bindings

The semantics for two-level data type bindings is given in Figure 52. As explained above, the first component returned is for specialization time data type bindings, and the second is for residual time data type bindings.

Two-Level Constructor Bindings

The semantics for two-level constructor bindings is given in Figure 53. The cases are straightforward.

Two-Level Exception Bindings

The semantics for two-level exception bindings is given in Figure 54. We return the binding in both static and dynamic version.

Two-Level Type Expressions

The semantics for two-level type expressions is given in Figure 55. The cases are straightforward.

Two-Level Type Expression Rows

The semantics for two-level type expression rows is given in Figure 56. The cases are straightforward.

$\mathcal{C}_{2valbind} : 2valbind \to seenlist \times valbind$

$\mathcal{C}_{2valbind}[\![2pat$ `=` $2exp\ \langle$`and` $2valbind\rangle]\!] \quad =$
  let $\langle$ (seenlist,$valbind$) $= \mathcal{C}_{2valbind}[\![2valbind]\!]\rangle$
  in ([]$\langle$ @ seenlist $\rangle$, $\mathcal{C}_{2pat}[\![2pat]\!]$ `=` $\mathcal{C}_{2exp}[\![2exp]\!]$ $\langle$`and` $valbind\rangle$)

$\mathcal{C}_{2valbind}[\![2pat\ \underline{=}\ 2exp\ \langle$`and` $2valbind\rangle]\!] \quad =$
  let $\langle$ (seenlist,$valbind$) $= \mathcal{C}_{2valbind}[\![2valbind]\!]\rangle$
  in ([]$\langle$ @ seenlist $\rangle$,
    `val _ =` $\lceil \mathcal{C}_{2pat}[\![2pat]\!]$ `=` $\mathcal{C}_{2exp}[\![2exp]\!] \rceil$ `;` $\langle valbind\rangle$)

$\mathcal{C}_{2valbind}[\![2pat\ \underline{\underline{=}}\ 2exp\ \langle$`and` $2valbind\rangle]\!] \quad =$
  let
    $\langle(seenlist, valbind) = \mathcal{C}_{2valbind}[\![2valbind]\!]\rangle$
    (`fn` var_1 `=>` ... `fn` var_n `=>` $2match$) $= 2exp \qquad (n \geq 0)$
    $var_{sp} = \mathcal{C}_{2pat}[\![2pat]\!]$
    $2mrule_1$ `|` ... `|` $2mrule_m = 2match \qquad (m \geq 2)$
    var_{do1} = mk_sp_do_fun (TypeOf(var_1),BTypeOf(var_1)
    $\vdots$
    var_{do_n} = mk_sp_do_fun (TypeOf(var_n),BTypeOf(var_n)
    var_{do_p} = mk_sp_do_fun (TypeOf($2match$),BTypeOf($2match$))
  in (`[(`var_{sp}`,`$\{1 \mapsto$ TypeOf(var_1), ...,
        $n \mapsto$ TypeOf(var_n), $(n+1) \mapsto$ TypeOf($2match$)$\}$`)]`
    $\langle$ @ $seenlist$ $\rangle$,
    var_{sp} `= fn` var_1 `=>` ... `fn` var_n `=> fn p =>`

```
let
  val (actualsres_1,gensplitexp_1,genseenB4_1,newvars_1) =
    var_do1 var_1
   ⋮
  val (actualsres_n,gensplitexp_n,genseenB4_n,newvars_n) =
    var_don var_n
  val (actualsres,gensplitexp,genseenB4,newvars) =
    var_dop p
  val abst_actualsres = concat [actualsres_1, ... ,actualsres_n]
  val abst_formalsres = concat [newvars_1, ... ,newvars_n]
  val formalsresexp_p = vars2exp abst_formalsres
  val formalsrespat_p = vars2pat abst_formalsres
```

Figure 42: Semantics for two-level value bindings

```
        val genseenB4s =
          (genseenB4_1, ... ,genseenB4_n,genseenB4)
        val (seen,name) =
          seenB4 (mk_seenB4list_name var_sp) genseenB4s
        val mk_sp_res_pat =
          include_mk_sp_res_pat abst_formalsres formalsrespat_p
        val sp_res_exp =
          include_mk_sp_res_exp actualsres abst_actualsres
        val _ =
          if seen then ()
          else emit name code
            (mk_sp_res_fn [
              let 2pat_1 = 2exp_1 = 2mrule_1
                (pat-pos-list_1,pat_1) = split_pat 2pat_1
              in
                Some(mk_mrule(mk_sp_res_pat pat-pos-list_1,
                  (fn pat_1=>C_2exp[[2exp_1]] handle x=>mk_exn x) gensplitexp))
                handle Match => None
              ⋮
              let 2pat_m = 2exp_m = 2mrule_m
                (pat-pos-list_m,pat_m) = split_pat 2pat_m
              in
                Some(mk_mrule(mk_sp_res_pat pat-pos-list_m,
                  (fn pat_m=>C_2exp[[2exp_m]] handle x=>mk_exn x) gensplitexp))
                handle Match => None
              ])
      in ⌈var_sp sp_res_exp⌉
      end
      ⟨valbind⟩)
```

$\mathcal{C}_{2valbind}[\![$`rec` $\mathit{2valbind}]\!] \quad =$

let $(\mathit{seenlist},\mathit{valbind}) = \mathcal{C}_{2valbind}[\![\mathit{2valbind}]\!]$

in $(\mathit{seenlist},$`rec` $\mathit{valbind})$

Figure 43: Semantics for two-level value bindings — continued

mk_sp_do_fun : Type × BType → *var*

mk_sp_do_fun(μ,τ) =
  if $((\mu,\tau), var_{do}) \in \sigma$ for some variable var_{do} then
    var_{do}
  else
    let var_{do} be fresh in
      $\sigma := \sigma \cup ((\tau,\beta), var_{do})$;
      if τ = `D` then
        v:=
          v `; and` var_{do} `= fn p => fn no =>`
            `let val newvar = fresh_var no in`
              `(p,⌈newvar⌉,blankcode,[newvar]) end`
      else case (μ,τ) of
          (`int`,S) =>
            v:= v `; and` var_{do} `= fn p => fn no =>`
                  `([],⌈p⌉,⌈p⌉,[])`
        | (`real`,S) => ... as for `int`
        | (`string`,S) => ... as for `int`
        | $(\{lab_1 \mapsto \mu_1, \ldots, lab_n \mapsto \mu_n\},$
          $\{lab_1 \mapsto \tau_1, \ldots, lab_n \mapsto \tau_n\})$ =>
          let var_{do_1} = mk_sp_do_fun(μ_1,τ_1)
              ⋮
              var_{do_n} = mk_sp_do_fun(μ_1,τ_1)
          in
            v :=
              v_n `; and` var_{do} `= fn p => fn no => let`
                `val (actualsres_1,gensplitexp_1,genseenB4_1,newvars_1) =`
                  var_{do_1} `(#`lab_1`⌈p⌉(no ^ "_" ^ "1")`
                ⋮
                `val (actualsres_n,gensplitexp_n,genseenB4_n,newvars_n) =`
                  var_{do_n} `(#`lab_n`⌈p⌉(no ^ "_" ^ "n")`
              `in (actualsres_1 @ ⋯ @ actualsres_n],`
                `{`lab_1 `= gensplitexp_1,...,`lab_n `= gensplitexp_n},`
                `{`lab_1 `= genseenB4_1,...,`lab_n `= genseenB4_n},`
                `newvars_1 @ ⋯ @ newvars_n])`
              `end`

Figure 44: The mk_sp_do_fun *operator*

| (t, S) => (* t is a type name, i.e., μ a constructed type *)
  let $(con_1, \ldots, con_n)$ = the constructors corresponding to t
    $\mu_i{-}opt$ = case Type(con_i) of $\mu \rightarrow t$ => Some μ,
                | t => None   for $i = 1..n$
    τ_i = BType(con_i),   for $i = 1..n$
    var_{do_i} = mk_sp_do_fun(μ_i, τ_i)
      for all $1 \leq i \leq n$ for which $\mu_i{-}opt$ = Some μ_i
  in

```
v := v_n ; and var_do =
(fn con_1 x => fn no =>        if μ_1-opt <> None
    let
      val (actualsres_1,gensplitexp_1,genseenB4_1,
           newvars_1) = var_do1 (no ^ "_" ^ "1")
    in
       (actualsres_1, con_1 gensplitexp_1,
        con_1 genseenB4_1,newvars_1)
    end
(fn con_1 => fn no =>        if μ_1-opt = None
    ([],con_1,con_1,[])
    ⋮
  | con_n x => fn no =>        if μ_n-opt <> None
    let
      val (actualsres_n,gensplitexp_n,genseenB4_n,
           newvars_n) = var_don (no ^ "_" ^ "n")
    in
       (actualsres_n, con_n gensplitexp_n,
        con_n genseenB4_n,newvars_n)
    end
  | con_n => fn no =>        if μ_n-opt = None
    ([],con_n,con_n,[])
end
```

return var_{do}

Figure 45: The mk_sp_do_fun *operator — continued*

$$\text{split\_atpat} : \mathit{2atpat} \to \mathit{string} \to (\mathit{pat} \times \mathit{string})\mathit{list} \times \mathit{atpat}$$

$$\begin{array}{lcl}
\text{split_atpat}\ [\![_]\!]\mathit{pos} & = & ([],_) \\
\text{split_atpat}\ [\![\mathit{scon}]\!]\mathit{pos} & = & ([],\mathit{scon}) \\
\text{split_atpat}\ [\![\mathit{var}]\!]\mathit{pos} & = & ([],\mathit{var}) \\
\text{split_atpat}\ [\![\mathit{con}]\!]\mathit{pos} & = & ([],\mathit{con}) \\
\text{split_atpat}\ [\![\mathit{excon}]\!]\mathit{pos} & = & ([],\mathit{excon}) \\
\text{split_atpat}\ [\![\{\langle \mathit{2patrow}\rangle\}]\!]\mathit{pos} & = & ([],\{\langle \text{split_patrow}\ [\![\mathit{2patrow}]\!]\mathit{pos}\ 1\rangle\}) \\
\text{split_atpat}\ [\![(\ \mathit{2pat}\ )]\!]\mathit{pos} & = & \text{split_pat}\ [\![\mathit{2pat}]\!]\mathit{pos}
\end{array}$$

Figure 46: The split_atpat *operator*

$$\text{split\_patrow} : \mathit{2patrow} \to \mathit{string} \to \mathit{int} \to (\mathit{pat} \times \mathit{string})\mathit{list} \times \mathit{patrow}$$

$$\begin{array}{l}
\text{split_patrow}\ [\![\mathit{lab}\ \texttt{=}\ \mathit{2pat}\ \langle\ \texttt{,}\ \mathit{2patrow}\rangle]\!]\mathit{pos}\ i \quad = \\
\quad \text{let}\ \langle(\mathit{pat\text{-}pos\text{-}list}', \mathit{patrow}') = \text{split_patrow}\ [\![\mathit{2patrow}]\!]\mathit{pos}\ (i+1)\rangle \\
\qquad (\mathit{pat\text{-}pos\text{-}list}, \mathit{pat}) = \text{split_pat}\ [\![\mathit{2pat}]\!](\mathit{pos}\ \hat{}\,(\text{makestring}\ i)) \\
\quad \text{in}\ (\mathit{pat\text{-}pos\text{-}list}\langle\ @\ \mathit{pat\text{-}pos\text{-}list}'\rangle, \\
\qquad \mathit{lab}\ \texttt{=}\ \mathit{pat}'\ \langle\ \texttt{,}\ \mathit{patrow}'\ \rangle
\end{array}$$

Figure 47: The split_patrow *operator*

$$\text{split\_pat} : \mathit{2pat} \to \mathit{string} \to (\mathit{pat} \times \mathit{string})\mathit{list} \times \mathit{pat}$$

$$\begin{array}{lcl}
\text{split_pat}\ [\![\mathit{2atpat}]\!]\mathit{pos} & = & \text{split_atpat}\ [\![\mathit{2atpat}]\!]\mathit{pos} \\
\text{split_pat}\ [\![\underline{\mathit{2atpat}}]\!]\mathit{pos} & = & ([(\phi(\mathit{2atpat}), \mathit{pos})], _) \\
\text{split_pat}\ [\![\mathit{con}\ \mathit{2atpat}]\!]\mathit{pos} & = & \\
\multicolumn{3}{l}{\quad \text{let}\ i = \text{the number for the constructor}\ \mathit{con}} \\
\multicolumn{3}{l}{\qquad (\mathit{pat\text{-}pos\text{-}list}, \mathit{atpat}) = \text{split_atpat}\ [\![\mathit{2atpat}]\!](\mathit{pos}\ \hat{}\,(\text{makestring}\ i))} \\
\multicolumn{3}{l}{\quad \text{in}\ (\mathit{pat\text{-}pos\text{-}list}, \mathit{con}\ \mathit{atpat})} \\
\text{split_pat}\ [\![\underline{\mathit{con}}\ \mathit{2atpat}]\!]\mathit{pos} & = & ([(\mathit{con}\ \phi(\mathit{2atpat}), \mathit{pos})], _) \\
\text{split_pat}\ [\![\mathit{excon}\ \mathit{2atpat}]\!]\mathit{pos} & = & (\text{* as normal constructors *}) \\
\text{split_pat}\ [\![\underline{\mathit{excon}}\ \mathit{2atpat}]\!]\mathit{pos} & = & ([(\mathit{excon}\ \phi(\mathit{2atpat}), \mathit{pos})], _) \\
\text{split_pat}\ [\![\mathit{var}\ \underline{\texttt{as}}\ \mathit{2pat}]\!]\mathit{pos} & = & ([(\mathit{var}\ \texttt{as}\ \phi(\mathit{2pat}), \mathit{pos})], _)
\end{array}$$

Figure 48: The split_pat *operator*

$$\mathcal{C}_{2atpat} : 2atpat \rightarrow atpat/\underline{atpat}$$

$$\begin{array}{lcl}
\mathcal{C}_{2atpat}[\![scon]\!] & = & scon \\
\mathcal{C}_{2atpat}[\![\underline{scon}]\!] & = & \lceil scon \rceil \\
\mathcal{C}_{2atpat}[\![var]\!] & = & var \\
\mathcal{C}_{2atpat}[\![\underline{var}]\!] & = & \lceil var \rceil \\
\mathcal{C}_{2atpat}[\![con]\!] & = & con \\
\mathcal{C}_{2atpat}[\![\underline{con}]\!] & = & \lceil con \rceil \\
\mathcal{C}_{2atpat}[\![excon]\!] & = & excon \\
\mathcal{C}_{2atpat}[\![\underline{excon}]\!] & = & \lceil excon \rceil \\
\mathcal{C}_{2atpat}[\![\{\langle 2patrow \rangle\}]\!] & = & \{\langle \mathcal{C}_{2patrow}[\![2patrow]\!] \rangle\} \\
\mathcal{C}_{2atpat}[\![\underline{\{}\langle 2patrow \rangle\underline{\}}]\!] & = & \lceil \{\langle \mathcal{C}_{2patrow}[\![2patrow]\!] \rangle\} \rceil \\
\mathcal{C}_{2atpat}[\![(\ 2pat\ )]\!] & = & (\mathcal{C}_{2pat}[\![2pat]\!]) \\
\mathcal{C}_{2atpat}[\![\underline{(}\ 2pat\ \underline{)}]\!] & = & \lceil (\mathcal{C}_{2pat}[\![2pat]\!]) \rceil
\end{array}$$

Figure 49: Semantics for two-level atomic patterns

$$\mathcal{C}_{2patrow} : 2patrow \rightarrow patrow/\underline{patrow}$$

$$\begin{array}{lcl}
\mathcal{C}_{2patrow}[\![lab = 2pat\ \langle\ ,\ 2patrow \rangle]\!] & = & lab = \mathcal{C}_{2pat}[\![2pat]\!]\ \langle\ ,\ \mathcal{C}_{2patrow}[\![2patrow]\!] \rangle \\
\mathcal{C}_{2patrow}[\![lab\ \underline{=}\ 2pat\ \langle\ \underline{,}\ 2patrow \rangle]\!] & = & \lceil lab = \mathcal{C}_{2pat}[\![2pat]\!]\ \langle\ ,\ \mathcal{C}_{2patrow}[\![2patrow]\!] \rangle \rceil
\end{array}$$

Figure 50: Semantics for two-level pattern rows

$$\mathcal{C}_{2pat} : 2pat \rightarrow pat/\underline{pat}$$

$$\begin{array}{lcl}
\mathcal{C}_{2pat}[\![2atpat]\!] & = & \mathcal{C}_{2atpat}[\![2atpat]\!] \\
\mathcal{C}_{2exp}[\![\underline{2atpat}]\!] & = & \lceil \mathcal{C}_{2atpat}[\![2atpat]\!] \rceil \\
\mathcal{C}_{2pat}[\![con\ 2atpat]\!] & = & con\ \mathcal{C}_{2atpat}[\![2atpat]\!] \\
\mathcal{C}_{2pat}[\![\underline{con}\ 2atpat]\!] & = & \lceil con\ \mathcal{C}_{2atpat}[\![2atpat]\!] \rceil \\
\mathcal{C}_{2pat}[\![excon\ 2atpat]\!] & = & excon\ \mathcal{C}_{2atpat}[\![2atpat]\!] \\
\mathcal{C}_{2pat}[\![\underline{excon}\ 2atpat]\!] & = & \lceil excon\ \mathcal{C}_{2atpat}[\![2atpat]\!] \rceil \\
\mathcal{C}_{2pat}[\![var\ \underline{\mathtt{as}}\ 2pat]\!] & = & \lceil var\ \mathtt{as}\ \mathcal{C}_{2pat}[\![2pat]\!] \rceil
\end{array}$$

Figure 51: Semantics for two-level patterns

$$\mathcal{C}_{2datbind} : \mathit{2datbind} \rightarrow \mathit{datbind}\ \mathit{Option} \times \underline{\mathit{datbind}}\ \mathit{Option}$$

$$\begin{array}{l}
\mathcal{C}_{2datbind}[\![\mathit{tycon}\ \texttt{=}\ \mathit{2conbind}\ \langle \texttt{and}\ \mathit{2datbind}\rangle]\!] \quad = \\
\quad \text{let}\ \langle\ (\mathit{datbind_opt},\mathit{datbind_opt}') = \mathcal{C}_{2datbind}[\![\mathit{2datbind}]\!]\ \rangle \\
\quad \text{in}\ (\text{Some}(\mathit{tycon}\ \texttt{=}\ \mathcal{C}_{2conbind}[\![\mathit{2conbind}]\!]\ \langle\ \text{case}\ \mathit{datbind_opt}\ \text{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{None} => \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mid \text{Some}\ \mathit{datbind} => \mathit{datbind}\ \rangle), \\
\qquad \text{None}\ \langle + \mathit{datbind_opt}'\ \rangle) \\
\mathcal{C}_{2datbind}[\![\mathit{tycon}\ \underline{\texttt{=}}\ \mathit{2conbind}\ \langle \texttt{and}\ \mathit{2datbind}\rangle]\!] \quad = \\
\quad \text{let}\ \langle\ (\mathit{datbind_opt},\mathit{datbind_opt}') = \mathcal{C}_{2datbind}[\![\mathit{2datbind}]\!]\ \rangle \\
\quad \text{in}\ (\text{None}\langle + \mathit{datbind_opt}\ \rangle, \\
\qquad \text{Some}(\lceil \mathit{tycon}\ \texttt{=}\ \underline{\mathcal{C}_{2conbind}[\![\mathit{2conbind}]\!]}\rceil\langle\ \text{case}\ \mathit{datbind_opt}\ \text{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{None} => \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\ \ \mid \text{Some}\ \mathit{datbind} => \mathit{datbind}\ \rangle)
\end{array}$$

Figure 52: Semantics for two-level data type bindings

$$\mathcal{C}_{2conbind} : \mathit{2conbind} \rightarrow \mathit{conbind}/\underline{\mathit{conbind}}$$

$$\begin{array}{l}
\mathcal{C}_{2conbind}[\![\mathit{con}\ \langle \texttt{of}\ \mathit{2ty}\rangle\ \langle\ \texttt{|}\ \mathit{2conbind}\rangle]\!] \quad = \\
\quad \mathit{con}\ \langle \texttt{of}\ \mathcal{C}_{2ty}[\![\mathit{2ty}]\!]\rangle\ \langle\ \texttt{|}\ \mathcal{C}_{2conbind}[\![\mathit{2conbind}]\!]\rangle \\
\mathcal{C}_{2conbind}[\![\underline{\mathit{con}}\ \langle \underline{\texttt{of}}\ \mathit{2ty}\rangle\ \langle\ \underline{\texttt{|}}\ \mathit{2conbind}\rangle]\!] \quad = \\
\quad \lceil \mathit{con}\ \langle \texttt{of}\ \underline{\mathcal{C}_{2ty}[\![\mathit{2ty}]\!]}\rangle\ \langle\ \texttt{|}\ \underline{\mathcal{C}_{2conbind}[\![\mathit{2conbind}]\!]}\rangle\rceil
\end{array}$$

Figure 53: Semantics for two-level constructor bindings

$$\mathcal{C}_{2exbind} : \mathit{2exbind} \rightarrow \mathit{exbind} \times \underline{\mathit{exbind}}$$

$$\begin{array}{l}
\mathcal{C}_{2exbind}[\![\mathit{excon}\ \langle \texttt{of}\ \mathit{2ty}\rangle\ \langle\!\langle \texttt{and}\ \mathit{2exbind}\rangle\!\rangle]\!] \quad = \\
\quad \text{let}\ \langle\!\langle (c,r) = \mathcal{C}_{2exbind}[\![\mathit{2exbind}]\!]\rangle\!\rangle \\
\quad \text{in}\ (\mathit{excon}\ \langle \texttt{of}\ \mathcal{C}_{2ty}[\![\mathit{2ty}]\!]\rangle\ \langle\!\langle \texttt{and}\ c\rangle\!\rangle, \lceil \mathit{excon}\ \langle \texttt{of}\ \underline{\mathcal{C}_{2ty}[\![\mathit{2ty}]\!]}\rangle\ \langle\!\langle \texttt{and}\ r\rangle\!\rangle\rceil) \\
\mathcal{C}_{2exbind}[\![\underline{\mathit{excon}}\ \langle \underline{\texttt{of}}\ \mathit{2ty}\rangle\ \langle\!\langle \texttt{and}\ \mathit{2exbind}\rangle\!\rangle]\!] \quad = \quad \mathcal{C}_{2exbind}[\![\mathit{excon}\ \langle \texttt{of}\ \mathit{2ty}\rangle\ \langle\!\langle \texttt{and}\ \mathit{2exbind}\rangle\!\rangle]\!]
\end{array}$$

Figure 54: Semantics for two-level exception bindings

$$
\begin{array}{lcl}
\mathcal{C}_{2ty} : \mathit{2ty} \rightarrow \mathit{ty}/\underline{\mathit{ty}} & & \\
\mathcal{C}_{2ty}[\![\texttt{\{}\langle \mathit{2tyrow}\rangle\texttt{\}}]\!] & = & \texttt{\{}\langle \mathcal{C}_{2tyrow}[\![\mathit{2tyrow}]\!]\rangle\texttt{\}} \\
\mathcal{C}_{2ty}[\![\underline{\texttt{\{}}\langle \mathit{2tyrow}\rangle\underline{\texttt{\}}}]\!] & = & \lceil\texttt{\{}\langle \underline{\mathcal{C}_{2tyrow}[\![\mathit{2tyrow}]\!]}\rangle\texttt{\}}\rceil \\
\mathcal{C}_{2ty}[\![\mathit{tycon}]\!] & = & \mathit{tycon} \\
\mathcal{C}_{2ty}[\![\underline{\mathit{tycon}}]\!] & = & \lceil\mathit{tycon}\rceil \\
\mathcal{C}_{2ty}[\![\mathit{2ty}\ \texttt{->}\ \mathit{2ty'}]\!] & = & \mathit{ty}\ \texttt{->}\ \mathit{ty} \\
\mathcal{C}_{2ty}[\![\mathit{2ty}\ \underline{\texttt{->}}\ \mathit{2ty'}]\!] & = & \lceil\mathit{ty}\ \texttt{->}\ \mathit{ty}\rceil \\
\mathcal{C}_{2ty}[\![\texttt{(}\ \mathit{2ty}\ \texttt{)}]\!] & = & \texttt{(}\ \mathit{ty}\ \texttt{)} \\
\mathcal{C}_{2ty}[\![\underline{\texttt{(}}\ \mathit{2ty}\ \underline{\texttt{)}}]\!] & = & \lceil\texttt{(}\ \mathit{ty}\ \texttt{)}\rceil
\end{array}
$$

Figure 55: Semantics for two-level type expressions

$$
\begin{array}{lcl}
\mathcal{C}_{2tyrow} : \mathit{2tyrow} \rightarrow \mathit{tyrow}/\underline{\mathit{tyrow}} & & \\
\mathcal{C}_{2tyrow}[\![\mathit{lab}\ \texttt{:}\ \mathit{2ty}\ \langle\ \texttt{,}\ \mathit{2tyrow}\rangle]\!] & = & \mathit{lab}\ \texttt{=}\ \mathcal{C}_{2ty}[\![\mathit{2ty}]\!]\ \langle\ \texttt{,}\ \mathcal{C}_{2tyrow}[\![\mathit{2tyrow}]\!]\rangle \\
\mathcal{C}_{2tyrow}[\![\mathit{lab}\ \underline{\texttt{:}}\ \mathit{2ty}\ \langle\ \underline{\texttt{,}}\ \mathit{2tyrow}\rangle]\!] & = & \lceil\mathit{lab}\ \texttt{=}\ \underline{\mathcal{C}_{2ty}[\![\mathit{2ty}]\!]}\ \langle\ \texttt{,}\ \underline{\mathcal{C}_{2tyrow}[\![\mathit{2tyrow}]\!]}\rangle\rceil
\end{array}
$$

Figure 56: Semantics for two-level type-expression rows

Two-Level Programs

The semantics for a two-level program *2dec* is given in Figure 57. The operator split_dec

$$
\begin{array}{l}
\mathcal{C}_{program} : \mathit{2dec} \rightarrow \mathit{dec} \\
\mathcal{C}_{program}[\![\mathit{2dec}]\!] \quad = \\
\quad \text{let } (\mathit{2dec}_1, \mathit{2dec}_2, \mathit{2dec}_3) = \text{split\_dec}[\![\mathit{2dec}]\!] \\
\qquad \mathit{dec}'_1 = \mathcal{C}_{2dec}[\![\mathit{2dec}_1]\!] \\
\qquad \mathit{dec}'_2 = \mathcal{C}_{2dec}[\![\mathit{2dec}_2]\!] \\
\qquad \mathit{dec}'_3 = \mathcal{C}_{2dec}[\![\mathit{2dec}_3]\!] \\
\quad \text{in} \\
\qquad \mathit{dec}_1; \\
\qquad \texttt{val}\ v; \\
\qquad \mathit{dec}_2; \\
\qquad \text{mk\_exn\_dec}(\mathit{2dec}_2) \\
\qquad \mathit{dec}_3
\end{array}
$$

Figure 57: Semantics for two-level type-expression rows

splits a two-level declaration into three parts. The first part contains all `datatype`-declarations, the second part contains all `exception`-declarations, and the third contains all value declarations.

Recall that the variable v is the value binding containing sp-do functions.

The operator mk_exn_dec is an operator that makes the declaration of the `mk_exn` function described in Section 3.9. Recall that the user program does not use the exception constructor `Match`, but possibly something else in its place. Let $excon_1$ range over all exception constructors without argument in the program *except* `Match`. Let further $excon_2$ range over all exception constructors with a dynamic argument, and let finally $excon_3$ range over all exception constructors with a static argument. The function `mk_exn` is defined as

```
fun mk_exn x =
  RAISE1exp
    (case x of
      excon1 => ATEXP1exp (EXCON1atexp "excon1")
      excon2 x => APP1exp (ATEXP1EXP (EXCON1atexp "excon2)",x)
      excon3 x => APP1exp (ATEXP1EXP (EXCON1atexp "excon3)",
                            LiftExp (TypeOf x) x)
    )
```

Note that if this function is called with the value `Match` then exception `Match` is raised. This is on purpose.

4.4 Correctness

In order to show the correctness of the two-level semantics, we must prove the following for any well-annotated two-level program *2dec* for which $dec = \phi(\mathit{2dec})$ is well-typed and terminates with the value v.

1. Consistency of static and dynamic two-level semantics, i.e., that the compiler generator does not "go wrong."

2. SML well-typedness of generating extension.

3. If the generating extension terminates, then it yields an SML well-typed residual program.

4. The residual program terminates with value v.

This is not an easy task due to the complexity of both the language and the techniques applied. An actual proof of this is outside the scope of this thesis.

Consistency proofs for a small expression language has been given by, among others, Milner [Mil78] and Tofte [Tof88]. A correctness proof for a partial evaluator for the lambda calculus has been given in [Gom92].

4.5 Summary

In this chapter we defined the grammar of the Two-Level Simplified Bare Language, and formally gave a static semantics expressing well-annotatedness requirements. Thereafter we formally specified the dynamic semantics of the Two-Level Simplified Bare Language. An implementation of the dynamic semantics corresponds to a compiler generator.

Chapter 5

Binding-Time Analysis

Come gather 'round people
wherever you roam
and admit that the waters
around you have grown
and accept it that soon
you'll be drenched to the bones
If your time to you is worth saving
then you'd better start swimming
or you'll sink like a stone
for the times, they are a-changing

— BOB DYLAN, *The Times are A-Changing (1963)*

In this chapter we present an efficient algorithm that produces a well-annotated two-level program from a one-level program. This process is known as binding-time analysis and is essential to off-line partial evaluation and to cogen.

The algorithm is based on the one presented in [Hen91] and which is also used as basis of for instance, [And92] and [BJ93]. For the first time we demonstrate that it is possible to use typedness of the source language to avoid doing an implicit standard type inference as part of the binding-time analysis.

Overview of The Algorithm

Given a well-typed ML program and the binding time of its main function the task is to find the binding times of all the phrases (declarations, expressions, et cetera) in the program. The *binding time* or *binding-time value* of a phrase describes how much and which parts of the value of the phrase that will be known at specialization time. Since the binding times control the production of the generating extension they should be as static as possible. The result of the algorithm is a well-annotated two-level program and the inferred binding times. Since there are many complex details in the following sections we start out by giving the reader the overall structure of the binding-time analysis algorithm.

- Between zero and three *binding-time variables* (β, $\overline{\beta}$, ...) are associated with every phrase in the entire program. Some of these binding-time variables will be used as place holders for the binding times, others will be used to express dependencies between variables.

- A set of binding-time *constraints* are systematically generated for the source program, describing the relations among the binding-time variables.

- A *solution* assigns binding-time values to all binding-time variables in such a way that all constraints are satisfied. There are always many different solutions. We desire *minimal* solutions, i.e., solutions that make as little as possible dynamic. A minimal solution is obtained by first *normalizing* the constraints. This results in a set of constraints for which the set of solutions is the same as for the original constraint set. From the normalized constraint set a minimal solution can easily be extracted.

- The source program is finally annotated to become a two-level program by a simple syntax-directed procedure using the inferred binding times.

5.1 Binding-Time Type Expressions

The binding-time type expressions (also known as binding-time values) are those generated by the following grammar:

$$\begin{array}{rcll} \tau & ::= & \mathtt{D} & (5.1) \\ & | & \mathtt{S} & (5.2) \\ & | & [\tau_1, \ldots, \tau_n] & (5.3) \end{array}$$

where $n \geq 0$. In words a value can be unknown, known base type, or structurally known. The reason for separating the latter two is somewhat technical. Identifying S with [ ] would lead to a minor change in the semantics of the lift predicate below (the unit value will be liftable) and it would require some minor changes in the proofs of the theorems in this chapter.

The notation $[\tau_1, \ldots, \tau_n]$ has nothing to do with lists, it is just convenient syntax. It is used for values having function type if the function can be applied at specialization time, record type if the components can be extracted at specialization time, or sum type if the constructor is known at specialization time. The context will always determine which.

Recall from Chapter 2 that the fields of records have been sorted and that the actual labels therefore have become irrelevant so there's enough information present. We expect a similar ordering on the constructors of a data type. When we know the context we will sometimes write binding-time values in a different way:

Notation	Representation	Remark
$\tau_1 \to \tau_2$	$[\tau_1, \tau_2]$	For function type values.
$\tau_1 * \cdots * \tau_n$	$[\tau_1, \ldots, \tau_n]$	For ML records, one component per label.
$\tau_1 + \cdots + \tau_n$	$[\tau_1, \ldots, \tau_n]$	For constructed values, one component per constructor of the data type.

(so in fact $\tau_1 \to \tau_2$ has the same representation as $\tau_1 + \tau_2$, but we will never again mix notation like this).

5.2 Constraint Systems

We introduce the convention that α's range over binding-time variables, that β's range over binding-time variables or the binding-time value $\mathtt{D}$, and that τ's range over binding-time values.

Definition 12 *A constraint system is a finite set of formal constraints, each of which must take one of the following forms:*

#	Constraint	Name
1	$\beta_1 \rightsquigarrow \beta_2$	lift constraint
2	$(\beta_1, \ldots, \beta_n) \triangleright \beta$, where $n \geq 0$	dependency constraint
3	$\beta_1 = \beta_2$	equality constraint
4	$[\beta_1, \ldots, \beta_n] \leq \beta$, where $n \geq 0$	structure constraint

Note that the notation here is just syntax and β's. There is no implied connection between say $[\beta_1, \ldots, \beta_n]$ and structured binding-time values (the interpretation is given below). Constraints of the form $(\beta_1) \triangleright \beta$ will often be written as $\beta_1 \triangleright \beta$ for short.

We use sets and not multi-sets for constraint systems unlike other authors, see [Hen91] and [And93]. We do this for readability reasons, but our arguments hold also for multi-sets.

To the above formal constraints we attach semantics.[1] Each constraint is either satisfied or not, so they are predicates in the binding-time variables. The semantics is relative to a substitution, S, that maps binding-time variables to binding-time values and has $S(\mathtt{D}) = \mathtt{D}$. Let τ_i stand for $S(\beta_i)$ (and so on) in the following.

- The equality predicate $\beta_1 = \beta_2$ is satisfied if and only if τ_1 and τ_2 consist of the same string of symbols.
- The dependency predicate $(\beta_1, \ldots, \beta_n) \triangleright \beta$ is satisfied if $\tau = \mathtt{D}$ or if $\tau_i \neq \mathtt{D}$ for some i. The informal meaning is that "if all these τ_i's are $\mathtt{D}$, then this τ must also be $\mathtt{D}$."

[1] Some authors insist on writing something like $\beta_1 \triangleright^? \beta_2$ for the formal constraints and $\beta_1 \triangleright \beta_2$ for the semantic constraints. This is out of step with hundreds of years of mathematical tradition (where no-one would write an inequality like $x * x \leq^? 4$) and quite unnecessary as misunderstandings are unlikely. And, by the way, if "$\leq$" and "$\leq^?$" are considered different, then so should the β's be; nevertheless they aren't.

- The structure constraint $[\beta_1, \ldots, \beta_n] \leq \beta$ is satisfied either by equality or if τ and all τ_i's are $\mathtt{D}$. In words, structured values that are dynamic have dynamic components.
- The lift predicate $\beta \rightsquigarrow \beta'$ is satisfied either by equality or if $\tau = \mathtt{S}$ and $\tau' = \mathtt{D}$. The lift constraint is used to describe the relation between two binding-time values in a situation where a `lift` might be inserted.

Example 4 Under the substitution $\{\beta_1 \mapsto \mathtt{S}, \beta_2 \mapsto \mathtt{D}, \beta_3 \mapsto [\mathtt{S},\mathtt{D}], \mathtt{D} \mapsto \mathtt{D}\}$ the following constraints are satisfied

$$\beta_1 \rightsquigarrow \mathtt{D} \qquad \beta_1 \rightsquigarrow \beta_2 \qquad [\beta_1, \beta_2] \leq \beta_3 \qquad (\beta_2, \beta_3) \triangleright \beta_1 \qquad (\mathtt{D}, \beta_2) \triangleright \beta_2 \qquad [\mathtt{D},\mathtt{D},\mathtt{D}] \leq \beta_2$$

while the following are not

$$\beta_2 \rightsquigarrow \beta_1 \qquad (\mathtt{D}, \beta_2) \triangleright \beta_1 \qquad [\beta_2, \beta_1] \leq \beta_3 \qquad [\mathtt{S},\mathtt{D},\mathtt{D}] \leq \beta_2 \qquad \beta_1 = \beta_2$$

We learn from this example that the order of components is important in structure constraints. □

5.2.1 Variable Equivalence

Definition 13 *For a constraint system, C, we define an equivalence relation on the set of binding-time variables and* $\mathtt{D}$ *as the smallest equivalence relation satisfying*

$$\beta_1 \approx \beta_2 \qquad \textit{if} \qquad \beta_1 = \beta_2 \in C \ \textit{or}\ \beta_1 \rightsquigarrow \beta_2 \in C$$

The reason for introducing this equivalence relation is that the lift predicate serves as a weakened equality predicate — variables that are equivalent with respect to this relation will most often turn out to be equal in the end.

5.2.2 Well-Typedness

We do not treat all kinds of constraint systems in our binding-time analysis; some cause trouble and since they are never generated from well-typed ML programs we disallow them. We require the constraint systems to satisfy the condition in the following definition.

Definition 14 (Well-typed constraint system) *Let C be a constraint system and let $R(C)$ be the subset of C consisting of all constraints not having the form $(\ldots) \triangleright \beta$. C is* well-typed *if $R(C)$ with all* $\mathtt{D}$*'s considered as different fresh variables can be solved equationally (i.e., if the inequalities and lifts can be replaced by $=$ and a most general unifier, $U(C)$, exists for the resulting equational system). Circular (or equivalently infinite) solutions are accepted.*

Remember that $\mathtt{D}$ is used for anything not known at specialization time — that is why it is treated as a wildcard in this definition. The exact purpose of this (admittedly rather obscure) definition is to make sure that whenever we have two structure constraints with the same right-hand side then they have the same number of components on the left-hand side. We will later sketch a proof that generated constraint systems are well-typed, and prove that the rewritings of constraint systems that we do all preserve well-typedness.

Example 5 Consider the following set of constraints

$$\{[\beta_1, \beta_2] \leq \beta_3, \quad [\beta_4, \beta_5, \beta_6] \leq \beta_3\} \tag{5.4}$$

This set violates definition 14 and thus is not well-typed. Intuitively the problem is that the set mixes values of different structure. □

5.2.3 Solutions

A constraint system is a kind of inequality system to be solved by replacing variables with values. To be specific

Definition 15 (Solutions to constraint systems) *Let C be a well-typed constraint system and let S be a substitution mapping all of C's binding-time variables into binding-time values. Let S' be the extension of S to all binding-time variables by mapping variables not in C to themselves and let S'' be the componentwise extension of S' to constraints also. We say that S is a solution of C if for every constraint $c \in C$ it holds that $S''(c)$ is satisfied.*

It is trivial to see that any constraint system has a solution, $f_{\mathtt{D}}$, which maps all variables present to the constant D. This property does not even depend on well-typedness — it is simply not possible to introduce semantic conflicts in a constraint system of the kind in question. We need more interesting solutions so we introduce the following partial ordering on solutions.

Definition 16 (Minimal solution) *We define a partial order on binding-time values by $\tau \preceq \mathtt{D}$, $\tau \preceq \tau$, and $[\tau_1, \ldots, \tau_n] \preceq [\tau'_1, \ldots, \tau'_n]$ if $\forall i : \tau_i \preceq \tau'_i$. We extend this to variables by defining $\alpha \preceq \alpha$ and $\alpha \preceq \mathtt{D}$. We finally extend it to solutions by defining $S \preceq S'$ if $\forall \alpha : S(\alpha) \preceq S'(\alpha)$. We then define*

$$S \prec S' \iff S \preceq S' \wedge S \neq S'$$

A solution S to a constraint system C is said to be minimal if there is no solution S' to C that satisfies $S' \prec S$.

Minimality for a solution means that it maps as few variables as possible to D. Some readers might notice that we have defined a weak minimality ("nothing is smaller") as opposed to a strong minimality ("less than anything else"). This is because our constraint systems may have solutions that are incommensurable as is the case for $\{\alpha_1 = \alpha_2\}$. Minimal solutions may map α to S, $[\mathtt{S}, \mathtt{S}]$ et cetera. The minimal solution we find in Theorem 7 is the first.

5.3 Normalizing the Constraints

This section describes the process of normalizing a set of constraints. The purpose of this process is to reduce the set to a form that can be solved essentially by "treating the inequalities as equalities." In order for this to succeed (equality-)conflicting constraints must be rewritten or eliminated. An example of possibly conflicting constraints is two inequalities with the same variable on the right-hand side and a dependency constraint:

$$\{[\beta_1, \beta_2] \leq \beta, \quad [\mathtt{D}, \mathtt{D}] \leq \beta, \quad \beta_1 \triangleright \beta\}$$

If we solve the first two constraints by equality then the last one fails.

The normalization process also eliminates non-trivial equality constraints. This is done by substitution. For example $\alpha = \mathtt{D}$ is eliminated by substituting $\mathtt{D}$ for α throughout the entire constraint system.

All reductions that we will make on constraint systems preserve solutions. This will be proved in Theorem 6.

Definition 17 (Normal form) *A normal form constraint system is a constraint system where only the following types of constraints occur*

- $\mathtt{D} = \mathtt{D}$.
- $\alpha \rightsquigarrow \beta$.
- $(\alpha_1, \ldots, \alpha_n) \triangleright \beta$, *where* $n \geq 1$.
- $[\beta_1, \ldots, \beta_n] \leq \alpha$, *where* $n \geq 0$.

and for which the following conditions hold

1. *If* $\alpha \approx \alpha'$ *then there are no two different* $\leq$ *constraints with* α *on the right-hand side of the first and* α' *on the right-hand side of the second.*
2. *If* $\alpha \approx \mathtt{D}$ *then* α *does not occur on the right-hand side of any* $\leq$ *constraint.*

We shall later see that the normal form condition is strong enough to let us solve a constraint system by equality yielding a minimal solution, and that every well-typed constraint system can be "normalized" in a solution preserving way.

5.3.1 Rewriting Rules

We will turn constraint systems into normal form by repeatedly applying the following rewriting rules as long as possible.

Substitution rules: some constraints involving equalities can be eliminated by applying a substitution to the set of constraints. These rules will be referred to as "rule SU-n." We will call the substitution in the right-hand column for the resulting substitution from applying a substitution rule.

#	Constraint	Substitution
1	$\alpha = \mathtt{D}$ or $\mathtt{D} = \alpha$	substitute $\mathtt{D}$ for α
2	$\alpha_1 = \alpha_2$	substitute α_1 for α_2

Simplification rules: some constraints can be expressed by other and simpler ones. These rules will be referred to as "rule SI-n." We will call the identity for the resulting substitution of applying a simplification rule.

#	Constraint	Replacement
1	$() \triangleright \beta$	$\beta = \mathtt{D}$
2	$(\beta_1, \ldots, \beta_{i-1}, \mathtt{D}, \beta_{i+1}, \ldots, \beta_n) \triangleright \beta$	$(\beta_1, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_n) \triangleright \beta$
3	$\mathtt{D} \rightsquigarrow \beta$	$\beta = \mathtt{D}$

Combination rules: some combinations of constraints can cause the system of constraints to be unsolvable as an equality system and must be rewritten. These rules will be referred to as "rule C-n." The column "Condition" contains preconditions that must be satisfied before the rules can be used. We will call the identity for the resulting substitution of applying a combination rule.

#	Constraints	Condition	Replacement
1	$[\beta_1, \ldots, \beta_n] \leq \alpha$, $[\beta'_1, \ldots, \beta'_n] \leq \alpha'$	$\alpha \approx \alpha' \not\approx \mathtt{D}$	$[\beta_1, \ldots, \beta_n] \leq \alpha$, $\beta_i = \beta'_i$, $\alpha = \alpha'$
2	$[\beta_1, \ldots, \beta_n] \leq \beta$	$\beta \approx \mathtt{D}$	$\beta_i = \mathtt{D}$, $\beta = \mathtt{D}$

This is the most important place where it can be seen that we don't do a standard type inference as part of the binding-time analysis. In for instance [Hen91] there are extra combination rules to deal with the possibility of say `S` and functional type values flowing together. Because the ML program and so its constraint system is well-typed, it is a priori known that similar things will not happen in our case. (We do not even allow `S` in constraint systems.) In [And92] there are also many combination rules, but that is because the author did not use the opportunity of using just one type of constraint for all the different source language types.

Definition 18 (The normalization process) *The normalization process is the process of exhaustive use of the above rewriting rules.*

The reader may very well wonder "why these rewriting rules and not some others?" — we both asked similar questions after reading other constraint rewriting rules. The key to the answer is that the normalized system of constraints must be solvable when the inequalities and lifts are regarded as equalities, because solving such a system is relatively easy.

The purpose of the substitution rules is to spread the information gathered and the purpose of the simplification rules is to eliminate equality conflicts found so far. Recall that structured values cannot be lifted and that variable equivalence between two variables is caused by a chain of lift (or equality) constraints connecting those variables. That means that lift constraints in connection with structure constraints can be regarded as equality constraints, and two structure constraints with the same right-hand side constitute a potential conflict.

5.3.2 Properties

We are now ready to prove a number of properties of constraint systems. Some of these are almost trivial and others are not, but they are all needed to understand why the algorithm we eventually present works.

Theorem 1 (Variable equivalence is preserved) *Let C be a well-typed constraint system and let C' be the resulting system after use of one of the rewriting rules. If $\beta_1 \approx \beta_2$ with respect to C, then $\tilde{\beta}_1 \approx \tilde{\beta}_2$ with respect to C' where $\tilde{\beta}_i$ is the resulting substitution applied to β_i.*

Proof Only the substitution rules and rule SI-3 remove any equality or lift constraint from C. In the substitution case the equivalence is moved into the substitution and rule SI-3 one equivalence causing constraint is replaced by another. In all other cases the theorem is trivially satisfied since the new relation is larger or equal. □

That we do not preserve equivalence by removing too much information in the rewriting process is a corollary of Theorem 6. In other words variable equivalence is not preserved trivially by collapsing the constraint system.

Theorem 2 (Well-typedness is preserved) *For every well-typed constraint system, C, any of the rewriting rules yields a well-typed constraint system, C'.*

Proof We prove this rule by rule:

For substitution rule 1 we have a simpler unification problem for $R(C')$ since different occurrences of β have been changed to different fresh variables. For substitution rule 2 we have $\alpha_1 = \alpha_2 \in R(C)$. This guarantees the existence of a $U(C')$.

Since all occurrences of `D`'s are replaced by different variables at unification time it is easy to see that all simplification rules preserve well-typedness.

For combination rule C-1 we have that $U(\alpha) = U(\alpha')$ since $\alpha \approx \alpha'$ and $\alpha \not\approx \mathtt{D}$. We then have $U(\beta_i) = U(\beta_i')$ so U is also a unifier for C'. For rule C-2 the unification problem in C' is simpler since the ith `D` matches anything that β_i matched. □

Again it follows from Theorem 6 that we do not obtain preservation by removing too much information.

Theorem 3 (Termination) *For any well-typed constraint system, C, the normalization process terminates.*

Proof Given any constraint system the number of possible consecutive substitution rules that can be applied is clearly bounded by the finite number of variables.

Now consider the sum $a+b$, where a is the number of constraints in the system and b is the total number of components on the left-hand sides of $\triangleright$ constraints. It is easily seen that an application of any rule, followed by as many applications of substitution rules as possible, reduces the sum and thus that the process terminates. □

Theorem 4 (Soundness) *Let C be a well-typed constraint system, and let C' be the constraint system obtained by normalization. C' is a normal form constraint system.*

Proof First we prove that only the allowed types of constraints occur in C'. Concerning lift constraints: the claim is that `D` does not occur on the left-hand side; this holds by rule SI-3. Concerning inequality constraints: the claim is that `D` does not occur on the right-hand side; this holds by rule C-2. Concerning dependency constraints: the claim is that `D` does not occur on the left-hand side and that the left-hand side is not empty; this holds by rules SI-1 and SI-2. Concerning equality constraint: there are none with variables by rules SU-1 and SU-2. This completes part one of the proof.

Second we prove that the claims 1–2 in the definition of normal form holds in C'. (1) holds by well-typedness (only compatible entities meet) and by rule C-1. (2) holds by rule C-2. □

Lemma 5 *Let C be a well-typed constraint system, let C' be the result of applying one of the rewriting rules on C and let U be the corresponding substitution (usually the identity function). Then*

$$S \text{ is a solution for } C \;\Rightarrow\; \exists S' : S = S' \circ U \quad \text{and} \quad S' \text{ is a solution for } C' \tag{5.5}$$

$$S' \text{ is a solution for } C' \;\Rightarrow\; S = S' \circ U \text{ is a solution for } C. \tag{5.6}$$

Proof Observe that the sets of variables used is unchanged except for the substitution rules and the requirement for substitutions to map variables not occurring in a constraint system to themselves is therefore preserved. Except for the substitution rules we further have that U is the identity substitution so that the above claims reduce to showing that solutions coincide.

We prove the two claims by examining the rewriting rules in turn.

- Substitution rule 1: The 1st claim is satisfied with $S' = V^{-1} \circ S \circ V$, where V maps α to a variable not used elsewhere and passes everything else. In words S' is S with the binding on α removed. S' maps α onto itself and no other variables disappear or arrive. The 2nd claim is satisfied since S must substitute `D` for α.

- Substitution rule 2: The 1st claim is satisfied as for the preceding rule, but with α_2 instead of α. The 2nd claim is satisfied since S must map α_1 and α_2 to the same thing.

- Simplification rules 1–3: these preserve solutions both ways as they are direct application of the semantics of constraints, see Section 5.2.

- Combination rule 1: The 1st claim holds since if $S(\alpha) = \mathtt{D}$ then also $S(\alpha') = \mathtt{D}$ and S clearly solves C'. Otherwise $S(\alpha) = [\tau_1, \ldots, \tau_n] = S(\alpha')$ since structured values cannot be lifted and because $\alpha \approx \alpha'$. 2nd claim: obvious.

- Combination rule 2: The 1st claim holds since $\beta \approx \mathtt{D}$ and because structured values cannot be lifted so we have $S(\beta) = \mathtt{D}$ and thus $S(\beta_i) = \mathtt{D}$. 2nd claim: obvious.

□

Theorem 6 (Solutions are preserved) *Let C be a well-typed constraint system, let C' be the result of applying one of the rewriting rules on C. Then there is a one-to-one correspondence between solutions of C and solutions of C'.*

Proof For non-substitution rules this was proved in the previous lemma. For the substitution rules we have the correspondence $S' = V^{-1} \circ S \circ V$ and $S = V \circ S' \circ V^{-1}$ using notation from the lemma. □

Theorem 7 (Minimal completions) *Every normal form constraint system C has a minimal solution.*

Proof Solve all the following constraints from C as an equality system:

- $[\beta_1, \ldots, \beta_n] \leq \alpha$.
- $\alpha \rightsquigarrow \alpha'$.

(i.e., leave out dependency constraints and lift constraints with D on the right-hand side.) This is possible due to well-typedness and the definition of normal form. Call the resulting substitution U and let T be the substitution that maps all variables in $U(C)$ to the constant value S. We claim that $V = T \circ U$ solves the entire system C and that V is a minimal solution.

Notice first that neither U nor T maps any variable to D. Every dependency constraint is therefore satisfied by V. For every $\alpha \rightsquigarrow$ D in C we have by the second condition of normal form that $V(\alpha) =$ S so the constraint is satisfied. All other remaining constraints are equality-solved by U and therefore satisfied by V. As variables not in C are mapped to themselves by both U and T we conclude that V is indeed a solution to C.

Now consider some other solution $V' \neq V$. If V' solves all the above mentioned constraints by equality it must map at least one of the remaining variables to something different from S and the two solutions are then either incommensurable or we have $V \preceq V'$. If on the other hand V' does not solve all the above mentioned constraints by equality then it maps some α to D where V does not, and we have $V' \not\preceq V$. We finally conclude that V is minimal. □

5.4 Comments on the Literature

The constraint solving algorithm presented in [Hen91] is for constraint systems similar to the ones presented here except that the only structures considered are functions. During our work we have found some mistakes in the article: using our notation, a constraint system like

$$\{\beta_1 \to \beta_2 \leq \beta_3, \quad \mathtt{D} \to \mathtt{D} \leq \beta_4, \quad \beta_5 \rightsquigarrow \beta_3, \quad \beta_5 \rightsquigarrow \beta_4, \quad \beta_1 \triangleright \beta_3\}$$

is a normal form constraint system in the sense defined in [Hen91] as none of the rewriting rules apply. The first four constraints in the system can be solved by equality but contrary to the claim in [Hen91, Proof 4] some variables are mapped to D. This invalidates the claim that dependency constraints are trivially satisfied. The same problem can be found in [JGS93, Section 8.7] and [And92, Chapter 5.2].

Basically the definition of a solution is the same in [Hen91] as in this paper; unused variables are required to be mapped to themselves. Strictly speaking this property together with Henglein's rewriting rule 3a invalidates the claim in [Hen91, Theorem 2] and its proof. To see this, one just has to consider the constraint system $\{\beta \triangleright \mathtt{D}\}$ which will be normalized to $\{\ \}$. The former system has a countable infinite number of solutions, the latter has exactly one solution. The same problem can be found in [JGS93, Section 8.7] and in [And92, Chapter 5].

Another point about [Hen91, Theorem 2] is that the claim ought to be sharpened: the claim is for the set of solutions but is in fact valid for the individual solutions (apart from the technical point above). The same point can be made about [JGS93, Proposition 8.4], [And92, Theorem 5.1], and [BJ93].

There are known problems with the actual code in [Hen91, Figure 4]. In particular a $\beta_1 \to \beta_2 \leq \mathtt{D}$ constraint may be left unprocessed in the leq-field of D.

5.5 Generating Constraints

In this section we describe the constraint system to generate for a well-typed Simplified Bare Language program. As it will be seen shortly, we use a large number of binding-time variables and generate a large number of constraints. On the other hand the number is linear in program size and the normalization is extremely efficient, so that is not a problem. For notational convenience (compactness to be precise) we use a slightly different notation for syntax trees in the figures. There is a one-to-one correspondence of the productions here and the productions of the Figures 4 and 5, so no problems should arise.

As we perform monovariant binding-time analysis we start out by assigning a binding-time variable β_v to every ML-variable v, a variable β_c to every constructor c, and a variable β_{tc} to every data type constructor in the program. It turns out that we need a little more information for variables in order to produce a well-annotated program, so we also assign another variable $\overline{\beta}_v$ to every v. This variable is called a flag as it is only allowed to take the values S and D. Flags are used to model the compile-time/residual-time (C/R) flags of the static semantics of the two-level language and this particular flag will be D if the variable is bound by a pattern that must be underlined and so will appear in the residual program. The C/R-combinator can be modeled by two dependency constraints: if $rc = rc_1 \sqcup rc_2$ then $\{\beta_{rc_1} \triangleright \beta_{rc}, \beta_{rc_2} \triangleright \beta_{rc}\}$ will work because binding-time variables default to S.

Figure 58 shows the generation of constraints for atomic expressions, expression rows, and expressions. We use one binding-time variable for atomic expressions to hold the inferred binding-time value for the phrase. For expression rows it turns out that there is no need for binding-time variables. To facilitate insertion of `lift`s when generating the two-level version of the expressions we use two binding-time variables to hold the inferred binding time for every expression, before (β_e) and after ($\overline{\beta}_e$) `lift`-insertion, and connect those with a lift-constraint. If a solution assigns different binding-time values to those variables it means that a `lift` must be inserted around the expression. As `lift` can never be inserted around a function or a raise expression, the lift-constraints there degenerate to equality constraints and one variable could be saved. It turns out to be simpler to

1	ae=`SCONatexp(_)`	$\{\}$
2	ae=`VARatexp(`v`)`	$\mathrm{varc}(\beta_{ae}, v)$
3	ae=`PARatexp(`e`)`	$\{\beta_{ae} = \overline{\beta}_e\}$
4	ae=`RECORDatexp(`er`)`	$\{\overline{\beta}_{e_1} * \cdots * \overline{\beta}_{e_n} \leq \beta_{ae}\}$
5	ae=`(EX)CONatexp(`c`)`	$\{\beta_{sum_c} = \beta_{ae}\}$, if c has no arguments
6	ae=`(EX)CONatexp(`c`)`	$\{\beta_c \to \beta_{sum_c} \leq \beta_{ae}, \beta_{sum_c} \triangleright \beta_{ae}\}$, if it has
7	ae=`LETatexp(`d, e`)`	$\{\beta_{ae} = \overline{\beta}_e, \beta_d \triangleright \beta_{ae}\}$
8	er=`EXPROW(_)`	$\{\}$
9	e=`ATEXPexp(`ae`)`	$\{\beta_e = \beta_{ae}, \beta_e \rightsquigarrow \overline{\beta}_e\}$
10	e=`APPexp(`e', ae`)`	$\{\beta_{e''} \to \beta_e \leq \overline{\beta}_{e'}, \beta_e \rightsquigarrow \overline{\beta}_e, \beta_{ae} \rightsquigarrow \beta_{e''}\}$
11	e=`FNexp(`m`)`	$\{\beta_e = \beta_m, \beta_e = \overline{\beta}_e, \overline{\beta}_e \triangleright \overline{\overline{\beta}}_e\} \cup \mathrm{spc}(e, m)$
12	e=`HANDLEexp(`e', m`)`	$\{\beta_e = \overline{\beta}_{e'}, \beta_{exn} \to \beta_e \leq \beta_m, \beta_e \rightsquigarrow \overline{\beta}_e\}$
13	e=`RAISEexp(`e'`)`	$\{\beta_e = \overline{\beta}_e\}$

Figure 58: Constraint generation for atomic expressions, expression rows, and expressions.

formulate the generation of constraints when both variables are present so we keep them. Note that the constraints generated for the application include the lift-constraint for the `ATEXPexp` that will be inserted around the argument during annotation.

As the source program is expected to be well-typed there is no need to generate constraints for constants — a minimal solution as the one described in Theorem 7 will always map β_{ae} to `S`.

The e_i's that appear in the rule for records are the expressions that the expression row contains. It is simpler to formulate the constraints there so the rule for expression rows is empty.

As explained in Section 3.8.1 pervasives must be treated polyvariantly and that is the reason for the varc function in the rule for variables. It is defined as

$$\mathrm{varc}(\beta_{ae}, v) = \begin{cases} \{\beta * \beta \leq \beta',\ \ \beta' \to \beta \leq \beta_{ae},\ \ \beta \triangleright \beta_{ae}\}, & \text{if } v \in \{+, -, \ldots\} \\ \{\beta * \beta \leq \beta',\ \ \beta' \to \beta_b \leq \beta_{ae},\ \ \beta \triangleright \beta_{ae},\ \ \beta_t + \beta_f \leq \beta_b\}, & \text{if } v \in \{=, <, \ldots\} \\ \{\beta \to \beta \leq \beta_{ae},\ \ \beta \triangleright \beta_{ae}\}, & \text{if } v \in \{\mathrm{ln}, \tilde{}, \ldots\} \\ \{\beta_v = \beta_{ae}\}, & \text{otherwise} \end{cases}$$

where the free β's except β_v defined above are new fresh variables for each occurrence of the pervasive functions. This ensures that monovariance is not enforced for pervasives.

The β_{sum_c} used in the rules for constructors is the binding-time variable assigned to the type to which the constructor belongs, i.e., its sum type. For exception constructors that type is type `exn`.

For the raise construct notice that no connection is made between the binding time of the raise value and the result. This is just like the standard static semantics of the raise construct: it does not affect the type of the result. The equality constraint is generated only to avoid erroneous `lift`s.

An extra flag variable $\overline{\overline{\beta}}_e$ is created for every `FNexp`. Its being `D` indicates that a specialization function must be inserted for the expression, either because the lambda is dynamic, or because the right branch cannot be chosen at specialization time. The flag is used in the utility function spc which is defined as

$$\mathrm{spc}(e, m) = \{\beta_p \to \beta_q \leq \beta_e, \overline{\overline{\beta}}_e \triangleright \beta_q\} \cup \mathrm{split}(m, \beta_p, \overline{\overline{\beta}}_e)$$

$$\cup \; \{\; \overline{\overline{\beta}}_e \triangleright \beta_v \;\mid\; v \in \mathrm{free}(e) \wedge \neg(\text{equality-type}(v)) \wedge \neg(\text{function-type}(v))\}$$

where β_p and β_q are binding-time variables not used elsewhere, free is a function that extracts the set of free variables in an expression. The idea is as follows. The $\leq$-constraint is used to split the binding-time variable for the whole expression into two binding-time variables, one for patterns in the match rule in the match (corresponding to the binding time of the argument to which the function potentially will be applied to) and one for the result. As explained $\overline{\overline{\beta}}_e$ will be `D` if we cannot determine which of the match rules in the match to choose given the binding time β_p of the patterns of the rules in the match, so the constraint $\overline{\overline{\beta}}_e \triangleright \beta_q$ simply expresses that a specialization function always returns dynamic values (code). The split function is used to determine whether or not a specialization point must be inserted; it is defined below. Finally, the set comprehension expresses that if a specialization function is to be inserted, then the variables to be abstracted over[2] must be dynamic if they are not of equality-type and not of function type.

Let us now consider how to define the split function. First of all, if there is only one rule in the match we certainly can decide which branch to choose, no matter what the binding time of the pattern is. Moreover, if there are n match rules we know that if we can choose among the first $n-1$ match rules then we can also choose among the n match rules; the reason is the requirement that matches be exhaustive: we are certain that if none of the $n-1$ match rules is to be chosen then the last rule must be chosen. So we are left with choosing among the first $n-1$ match rules. If there in one of these match rules is a constant and the corresponding binding time is `D`, then we cannot determine which branch to choose and the flag $\overline{\overline{\beta}}_e$ must be set to `D`. Consider the following example where we have three match rules:

```
fn (1, x)      => ...
 | (2, (1,3)) => ...
 | _           => ...
```

Assume that the binding time for the patterns is $\mathtt{S} * (\mathtt{S} * \mathtt{D})$. Then we cannot determine which branch to choose as the binding time corresponding to the constant `3` is `D`. So the idea will be to decompose the binding time for the patterns according to the patterns and generate $\cdot \triangleright \overline{\overline{\beta}}_e$ constraints for all constants. As it can be seen from the example, it is not obvious how to decompose the binding time for the patterns — the second pattern would intuitively yield a more fine-grained decomposition than the first. The crucial insight is, however, that it is not needed to find *one* most fine-grained decomposition. We can instead consider each pattern in turn, decomposing the binding time each time! It is not only for special constants that $(\cdot) \triangleright \overline{\overline{\beta}}_e$ constraints should be generated. If the pattern is a a constructed pattern then such a constraint should also be generated with the binding-time variable for the constructed pattern substituted for the dot; because if we do not

[2]This was discussed in Section 3.1.5.

know the constructor, we cannot determine the branch (constructors correspond here to special constants).

Hence split is defined as follows. For each pattern but the last in the match rules of the match apply the function split_{pat} in figure 59 below to β_p and $\overline{\overline{\beta}}_e$ (recall that β_p is binding-time variable for the pattern). This completes the description of the generated constraints for the various kinds of expressions and we now go on to consider the other phrase classes.

$$
\begin{array}{l}
\mathrm{split}_{pat}(p, \beta_p, \overline{\overline{\beta}}_e) = \\
\quad \text{case } p \text{ of} \\
\qquad \quad \texttt{ATPATpat}(ap) \Rightarrow \mathrm{split}_{atpat}(ap, \beta_{ap}, \overline{\overline{\beta}}_e) \\
\qquad \mid \texttt{(EX)CONpat}(c, ap) \Rightarrow \{\beta_p \triangleright \overline{\overline{\beta}}_e\} \cup \mathrm{split}_{atpat}(ap, \beta_{ap}, \overline{\overline{\beta}}_e) \\
\qquad \mid \texttt{LAYEREDpat}(v, p) \Rightarrow \mathrm{split}_{pat}(p, \beta_p, \overline{\overline{\beta}}_e) \\
\\
\mathrm{split}_{atpat}(ap, \beta_p, \overline{\overline{\beta}}_e) = \\
\quad \text{case } p \text{ of} \\
\qquad \quad \texttt{SCONatpat}(\_) \Rightarrow \{\beta_p \triangleright \overline{\overline{\beta}}_e\} \\
\qquad \mid \texttt{VARatpat}(v) \Rightarrow \{\} \\
\qquad \mid \texttt{(EX)CONatpat}(\_) \Rightarrow \{\beta_p \triangleright \overline{\overline{\beta}}_e\} \\
\qquad \mid \texttt{RECORDatpat}(l_i = p_i) \Rightarrow \qquad \text{where } i \in \{1, \ldots, n\} \\
\qquad \qquad \text{let } \beta_{p_i} = \beta, \qquad \text{where } \beta \text{ is fresh} \\
\qquad \qquad \text{in } \{\beta_1 * \cdots * \beta_n \leq \beta_p\} \cup \bigcup_i (\mathrm{split}_{pat}(p_i, \beta_i, \overline{\overline{\beta}}_e))
\end{array}
$$

Figure 59: Algorithms split_{pat} *and* split_{atpat}

Figure 60 shows the generation of constraints for matches and match rules. We employ

1	ma=`MATCH`(mr)	$\{\beta_{ma} = \beta_{mr}\}$
2	ma=`MATCH`(mr, ma')	$\{\beta_{ma} = \beta_{mr}, \beta_{ma} = \beta_{ma'}\}$
3	mr=`MRULE`(p, e)	$\{\beta_p \to \overline{\beta}_e \leq \beta_{mr}, \beta_{mr} \triangleright \overline{\beta}_p\}$

Figure 60: Constraint generation for matches and match rules.

one binding-time variable for each match and one for each match rule, in both cases to hold the inferred binding time which corresponds to the binding time of the lambda expression they are part of. The generation is quite straightforward, at least compared to the expressions; the constraints for matches ensure that all match rules in a match share the same binding time and the constraints for match rules make the binding time for a match rule (and thus for a lambda) equal to a function from the binding time of the pattern to the binding time of the expression, *unless* they are all `D`.

Figure 61 shows the generation of constraints for atomic patterns, pattern rows, and patterns. We assign a binding-time variable β_p to all (atomic) patterns corresponding

1	p=`ATPATpat`(ap)	$\{\beta_p = \beta_{ap}, \overline{\beta}_p \triangleright \overline{\beta}_{ap}, (\overline{\overline{\beta}}_e, \beta_p) \triangleright \overline{\beta}_p\}$
2	p=`(EX)CONpat`(c, ap)	$\{\beta sum_c = \beta_p, \beta_c = \beta_{ap}, \beta_p \triangleright \overline{\beta}_p, \overline{\beta}_p \triangleright \overline{\beta}_{ap}, (\overline{\overline{\beta}}_e, \beta_p) \triangleright \overline{\beta}_p, \overline{\beta}_{ap} \triangleright \overline{\beta}_e\}$
3	p=`LAYEREDpat`(v, p')	$\{\overline{\beta}_v \triangleright \beta_v, \overline{\beta}_v = \mathtt{D}, \overline{\beta}_p \triangleright \beta_p, \overline{\beta}_p = \mathtt{D}\}$
4	ap=`PARatpat`(p)	$\{\beta_{ap} = \beta_p, \overline{\beta}_{ap} \triangleright \overline{\beta}_p\}$
5	ap=`SCONatpat`$(\_)$	$\{\overline{\beta}_{ap} \triangleright \beta_{ap}\}$
6	ap=`VARatpat`(v)	$\{\beta_{ap} = \beta_v, \overline{\beta}_{ap} \triangleright \beta_{ap}, \overline{\beta}_{ap} \triangleright \overline{\beta}_v\}$
7	ap=`RECORDatpat`(pr)	$\{\beta_{p_1} * \cdots * \beta_{p_n} \leq \beta_{ap}, \overline{\beta}_{ap} \triangleright \overline{\beta}_{p_i}, \beta_{ap} \triangleright \overline{\beta}_{ap}, \overline{\beta}_{ap} \triangleright \overline{\beta}_e\}$
8	ap=`(EX)CONatpat`(c)	$\{\beta sum_c = \beta_{ap}, \beta_{ap} \triangleright \overline{\beta}_{ap}\}$
9	pr=`PATROW`$(\_)$	$\{\}$

Figure 61: Constraint generation for atomic patterns, pattern rows, and patterns.

to the value it is matched against. Furthermore we assign a flag $\overline{\beta}_p$ to the pattern. The flag which is used to model the *rc*-flag of the static semantics is `D` when the pattern is to be underlined. When this happens in a lambda expression the whole expression must by dynamic and that explains the last constraint for record and constructed patterns. In those constraints e stands for the lambda expression which the pattern is part of. The pattern cannot be part of an exception handling or the left-hand side of a value binding due to the syntactic restrictions for the Simplified Bare Language .

As the variable rule shows, variables that are bound by underlined patterns are themselves underlined.

As with expression rows, the constraints for pattern rows are easier formulated at the atomic pattern level. The p_i's that appear in the rule for record atomic patterns are the patterns of the pattern row. Consequently the set generated for pattern rows is empty. Again as with the expression rules, βsum_c appears in the constructor rules and is the binding-time variable associated with the corresponding type. For exceptions that type is of course type `exn`.

As discussed in Section 3.6.2 layered patterns are always made residual. We use two dependency constraints instead of equality constraints even though they will immediately reduce to equality constraints. This is done to ease the proof of Theorem 8.

Figure 62 shows the generation of constraints for declarations and bindings. We use one binding-time variable for declarations and for value bindings; they model the *rc*-flags of the static semantics. We use no binding-time variables for data type, constructor, and exception bindings. The relevant information is present in the variables for the value, exception and type constructors.

The c_i's that appear in the rule for data type bindings are the constructors that are bound. This situation is similar to the record expressions et cetera. Note also that the β_{tc} is the binding-time variable for the sum type, i.e., if c is a constructor of this sum type, then $\beta sum_c = \beta_{tc}$. This may introduce recursive types.

When there is no argument to a data type constructor (rule 9) or to an exception

1	d=`SEQdec`(d_1, d_2)	$\{\beta_{d_1} \triangleright \beta_d, \beta_{d_2} \triangleright \beta_d\}$
2	d=`EMPTYdec`$()$	$\{\}$
3	d=`VALdec`(vb)	$\{\beta_{vb} \triangleright \beta_d\}$
4	d=`DATATYPEdec`(db)	$\{\}$
5	d=`EXCEPTIONdec`(eb)	$\{\}$
6	vb=`PLAINvalbind`(p, e, vb')	$\{\beta_p = \overline{\beta}_e, \beta_p \triangleright \beta_{vb}, \beta_{vb'} \triangleright \beta_{vb}\}$
7	vb=`RECvalbind`(vb')	$\{\beta_{vb} = \beta_{vb'}\}$
8	db=`DATBIND`(tc, cb, db')	$\{\beta_{c_1} + \cdots + \beta_{c_n} \leq \beta_{tc}\}$
9	cb=`CONBIND`(c, cb')	$\{\}$
10	cb=`CONBIND`(c, ty, cb')	$\{\beta_c = \beta_{ty}\}$
11	eb=`EXBIND`(c, eb')	$\{\}$
12	eb=`EXBIND`(c, ty, eb')	$\{\beta_c \rightsquigarrow \mathtt{D}, \beta_c = \beta_{ty}\}$

Figure 62: Constraint generation for declarations and bindings.

constructor (rule 11), a different set of constraints is generated than otherwise. The lift constraint in the last rule assures that the argument can be lifted in case an exception occurs in a non-strict context and the exception package therefore must be lifted.

Figure 63 shows the generation of constraints for types and type rows. Just as for

1	ty=`RECORDty`(tr)	$\{\beta_{ty_1} * \cdots * \beta_{ty_n} \leq \beta_{ty}\}$
2	ty=`CONty`$(\_, tc)$	$\{\beta_{sum_{tc}} = \beta_{ty}\}$
3	ty=`FNty`(ty_1, ty_2)	$\{\beta_{ty_1} \rightarrow \beta_{ty_2} \leq \beta_{ty}\}$
4	ty=`PARty`(ty')	$\{\beta_{ty'} = \beta_{ty}\}$
5	tr=`TYROW`$(\_)$	$\{\}$

Figure 63: Constraint generation for types and type rows.

expression rows and pattern rows, the constraints associated with record types are moved from the type rows to the record type, which then becomes empty.

The $\beta_{sum_{tc}}$ in the rule for constructed type is the binding-time variable associated with the type constructor.

In addition to these constraints some "external constraints" are generated to set the binding time of the main function we want to specialize. If for instance the user declares that the curried function `ack` has two parameters, the first being static and the second dynamic, then we generate

$$\{\beta_1 \rightarrow \beta_1' \leq \beta_{ack}, \quad \beta_2 \rightarrow \beta_2' \leq \beta_1', \quad \beta_2 = \mathtt{D}, \quad \beta_2' = \mathtt{D}\}$$

to put the start information, $\beta_{ack} = \mathtt{S} \rightarrow (\mathtt{D} \rightarrow \mathtt{D})$, into the system.

Theorem 8 *The constraint system generated for a well-typed monomorphic SML program by the rules in Section 5.5 is well-typed.*

Proof (Sketch) A well-typed program can be type checked and types can be assigned to every subexpression and every identifier in the program. We have assumed programs to be monomorphic, so this can be done without type variables.

When we remove all $(\ldots) \triangleright \beta$ constraints from the generated set of constraints, few flag variables survive. Those that do survive participate only in equality constraints with other flag variables and will therefore not interfere when we unify the other variables as the well-typedness definition (14) requires.

We will now show that a most general unifier exists for the reduced constraint system by exhibiting one unifier. The unifier we exhibit does not assign binding-time types to binding-time variables as one would expect — it assigns Standard ML types! More precise, we assign all non-flag binding-time variables the ML type of the corresponding phrase. Because the types are variable free, this unifies the constraint system. $\square$

5.6 Program Annotation

We will now describe how to obtain a well-annotated two-level program from a one-level program, using the constraint system generated by the algorithm in the previous section and its solution. This is a five-phase algorithm (which however can be implemented by one pass over the syntax tree): syntax modification, lift-insertion, sp-function insertion, underlining, and dynamic pervasives corrections.

- First of all the application syntax is changed slightly from the one-level syntax to the two-level syntax (as described in Section 4.1.2) so we insert an `ATEXPexp` tag around every application argument. The new expressions arising are assigned the two binding-time variables β_{ae} and $\beta_{e''}$ from Figure 58.

- Second, specialization functions (aka sp-functions) are inserted for every `FNexp` having $\overline{\overline{\beta}}_e = \texttt{D}$. The sp-functions must be inserted according to the rules in section 3.1.5. The original lambda expression is replaced by a call to the sp-function with the abstracted variables as parameters. The lambda expressions and application introduced due to abstraction need not be underlined by the fourth phase.

- Third, we insert `lift`s around every expression for which the solution has assigned different binding-time values for the two variables. This includes the expressions introduced in the first phase.

- In the fourth phase we insert underlinings using the following rules. The β's referenced are those in the figures describing the generation of constraints.

  Atomic expressions: An `SCONatexp` is never underlined, a `VARatexp` is underlined if $\beta_v = \texttt{D}$; it is underlined twice if $\overline{\beta}_v = \texttt{D}$ and it is not bound by an abstraction caused by sp-function insertion, an `(EX)CONatexp` is underlined if $\beta_{ae} = \texttt{D}$, a `RECORDatexp` is underlined if $\beta_{ae} = \texttt{D}$, a `LETatexp` is underlined if $\beta_d = \texttt{D}$, a `PARatexp` is underlined if the expression below is underlined (`lift`s are considered underlined for this purpose).

Expression rows: Underlined when the corresponding `RECORDatexp` is.

Expressions: An `ATEXPexp` is underlined when the atomic expression below is underlined, a `APPexp` is underlined if $\overline{\beta}_{e'} = \mathtt{D}$, a `FNexp` is underlined is $\beta_e = \mathtt{D}$, a `HANDLEexp` is underlined if the inner expression is (again `lift`s are considered underlined), a `RAISEexp` is underlined if $\beta_{e'} = \mathtt{D}$.

Matches: Underlined if $\beta_m = \mathtt{D}$.

Match rules: Underlined if $\beta_{mr} = \mathtt{D}$.

Patterns: Underlined if $\overline{\beta}_p = \mathtt{D}$.

Atomic patterns: Underlined if $\overline{\beta}_p = \mathtt{D}$.

Pattern rows: Underlined if the corresponding `RECORDatpat` is.

Declarations: Never underlined.

Value bindings: A `PLAINvalbind` is underlined once if $\overline{\beta}_e = \mathtt{D}$ and twice if it binds a specialization function, a `RECvalbind` is never underlined.

Data type bindings: Never underlined.

Constructor binding: Underlined if $\beta_{sum_c} = \mathtt{D}$.

Exception bindings: Underlined if $\beta_{exn} = \mathtt{D}$.

Types: Underlined if $\beta_{ty} = \mathtt{D}$.

Type rows: Underlined if the corresponding `RECORDty` is.

- Finally underlined pervasives, $\underline{op}$, are replaced by `(mk_op op)` to get duovariance for pervasives. (We cannot use the name `+` for both the static and the dynamic case in the generating extension.)

Theorem 9 *The two-level programs generated by the above algorithm are well-annotated in the sense defined in Section 4.2.3.*

Proof By a tedious comparison of the constraints generated, the annotations rules, and the rules of in Section 4.2. □

5.7 Efficient Implementation

A naïve implementation of the normalization process described above would be very time consuming since it would require searching for some constraints every time one of the combination rules should be used. This section presents an efficient implementation that runs in almost linear time.

The key to efficiency is to present the constraints one by one and to record enough information about already-seen constraints to make it possible to decide which rules can be applied when a new constraint is presented.

Phase 1 — initialization

- Apply simplification rules SI-1 and SI-2 exhaustively. As nobody in his/her right mind would generate constraints requiring such rewriting, no real action is required.
- For every $\beta_1 \rightsquigarrow \beta_2$ constraint add a $\beta_1 \triangleright \beta_2$ constraint. This handles rule SI-3 in a simple way.
- Using union-find techniques an evolving equivalence relation can be constructed on the set of binding-time variables and D. This relation is used to trace variable substitutions (*not* variable equivalence!) The class in which β occurs is written $\tilde{\beta}$ and we associate a set of dependencies, $\mathrm{dps}(\tilde{\beta}) \equiv \mathrm{dps}(\beta)$, with every class.
- For each constraint $(\beta_1, \ldots, \beta_k) \triangleright \beta$ create a pair (k, β). Insert a reference to this pair in all $\mathrm{dps}(\beta_i)$. Since k may be decremented later it is important that all the β_i's reference the same pair in memory. The meaning of such a pair is that k is decremented every time some β_i turns D and if k eventually gets decremented to 0 then $\beta = \mathtt{D}$ will be inserted into the working list.
- Initialize the working list to all non-$\triangleright$ constraints. A copy of this list must be kept for phase 3.
- Construct a term graph for all members of the working list using nodes with labels S (arity 0), D (arity 0), $=$ (arity 2), $\rightsquigarrow$ (arity 2), $[\,]$ (some finite arity), and $\leq$ (arity 2). The binding-time variables are the other leaves of the graph.
- On the set of classes, $\tilde{\beta}$, create another evolving equivalence relation for handling variable equivalence. Call the resulting classes $\tilde{\tilde{\beta}}$. With each $\tilde{\tilde{\beta}}$ we associate a field, $\mathrm{memory}(\tilde{\beta}) \equiv \mathrm{memory}(\beta)$, which is to hold at most one unprocessed inequality constraint with β on the right-hand side. Notice by inspection of the combination rules that it does not matter which $\beta \in \tilde{\beta}$ or $\tilde{\beta} \in \tilde{\tilde{\beta}}$ is used. The memory field of D will always be kept empty.

Figure 64 shows a snapshot of the part of the dependency graph that represents the constraint $(\beta_{42}, \beta_{43}, \beta_{44}, \beta_{45}) \triangleright \beta_{99}$ after seeing the constraints $\beta_{42} = \beta_{44}$ and $\beta_{45} = \mathtt{D}$. Note that if β_{42} turns D then the counter will be decremented twice.

Phase 2 — main While the worklist is not empty remove a constraint, c, from it. If there is a lift or an equality constraint in the list, c must be such a constraint. (In other words we handle inequalities only when no other constraints are available.)

If c is an equality or a lift constraint we now recognize the two sides to be variable equivalent and therefore unify the two $\tilde{\tilde{\beta}}$ classes if they are not already equivalent. This may lead to application of one of the combination rules since two conflicting memory fields may collide or one of c's sides may be $\approx$-equivalent to D.

If c is an equality constraint with different sides we also unify the two $\tilde{\beta}$ classes thus recording the substitution. If both sides are variables then the dps-sets are concatenated. If on the other hand one of the sides is D we update the dps-sets as described above.

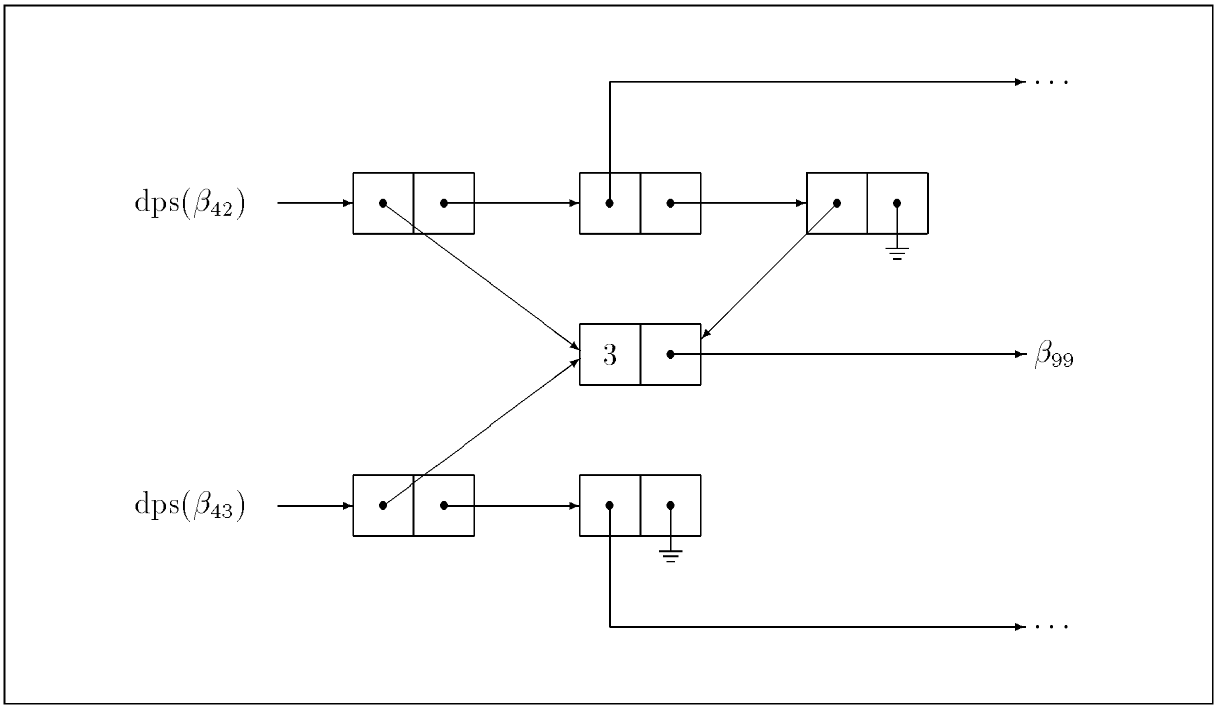

Figure 64: Snapshot of dependency constraint graph.

If c is a structural constraint with right-hand side β we have by the choice of c that the $\approx$-relation is up to date. If there is no remembered constraint in memory(β) simply record c there. Otherwise we apply one of the combination rules.

Phase 3 — postprocessing

- For each $(\ldots) \leq \beta$ or $\cdot \rightsquigarrow \beta$ in the saved copy of the working list solve by equality if $\tilde{\beta} \neq \tilde{\mathtt{D}}$. This is done efficiently by changing β to a link to the left-hand side when the sides are different.
- Unify any remaining free variable with `S`.

After phase 3 all nodes representing binding-time variables have been unified with a node representing their binding-time values — the process is complete.

Actual code Figures 65–67 show the code needed for normalization. For readability we have used standard list notation for the dependency lists although our complexity analysis requires that the append operation on lists is $O(1)$ in time complexity. To ensure this a special data type with direct access to both ends of the list should be used.

Complexity The algorithm described above is very efficient and our implementation confirms this. Without going into details we here deduce its complexity as a function of the number of components in the constrain set. The number of components is defined as the number of constraints plus the number of occurrences of variables and constants.

```
datatype BType = (* Graph Nodes *)
    STATIC
  | DYNAMIC of BLType ref
  | CONSTR of BType ref list
  | BTVAR of (int ref*Btype ref) list ref * BLType ref
  | BTLINK of BType ref

and BLType = (* Equivalences on DYNAMIC/BTVAR *)
    BTLEQR of (BType ref*BType ref) Option ref
  | BTLLINK of BLType ref

and Constraint = (* Constraints in worklist *)
    EQUAL of BType ref*BType ref
  | LIFT of BType ref*BType ref
  | LEQ of BType ref*BType ref

val Snode = ref STATIC
val DLnode = ref (BTLEQR (ref None))
val Dnode = ref (DYNAMIC DLnode)
val worklist = ref ([] : Constraint list)
val btvarlist = ref ([] : BType ref list)

fun find  n  = ... (* Follow BTLINKs using transition halving *)
fun findL nl = ... (* Follow BTLLINKs using transition halving *)
fun isD n   = (find n = Dnode)
fun isDL nl = (findL nl = DLnode)
fun isvar (BTVAR _) = true
  | isvar _ = false

fun geteqr (ref (DYNAMIC eqr)) = eqr
  | geteqr (ref (BTVAR (dps,eqr))) = findL eqr
  | geteqr _ = crash ()

fun getmem (ref (BTLEQR mem)) = mem
  | getmem _ = crash ()

fun apply f xs = (map f xs; ())
fun mk_eqconstraint c = worklist := EQUAL c :: (!worklist)
fun update_dps (k,n) =
  (k := !k - 1; if !k = 0 then mk_eqconstraint (n,Dnode) else ())
```

Figure 65: Normalization algorithm — part one.

We assume that removal (insertion) of constraints from (into) the working list, addition of dps-sets, and similar simple operations are performed in constant time. This can be achieved by suitable data structures. Even though our implementation does not satisfy these conditions it is still very fast, but we suspect that it spends most of its time appending lists.

The total number of non-= constraints processed is bounded by twice the number of constraints. The total number of = constraints processed is bounded by the number of constraints plus the number of components on the left-hand sides of constraints.

```
fun doC1 (l1,r1) (l2,r2) = (* Combination rule 1 *)
  case (!(find l1),!(find l2)) of
    (CONSTR c1,CONSTR c2) =>
      apply mk_eqconstraint (zip (r1 :: c1) (r2 :: c2))
  | _ => crash ()

fun doC2 (l,r) = (* Combination rule 2 *)
  case !(find l) of
    (CONSTR c) =>
      apply (fn n => mk_eqconstraint (n,Dnode)) (r :: c)
  | _ => crash ()

fun unionL n1 n2 = (* Make nodes var-equivalent *)
  let val (btl1,btl2) = (geteqr bt1,geteqr bt2)
  in if !btl1 = !btl2 then () else
    case (!btl1,!btl2) of
      (BTLEQR (ref None),BTLEQR (ref None)) =>
        if isDL(btl1) then btl2 := BTLLINK btl1
                      else btl1 := BTLLINK btl2
    | (BTLEQR (ref None),BTLEQR (ref (Some c))) =>
        if isDL btl1 then doC3 c; btl2 := BTLLINK btl1
                     else btl1 := BTLLINK btl2
    | (BTLEQR (ref (Some c)),BTLEQR (ref None)) =>
        if isDL btl2 then doC3 c; btl1 := BTLLINK btl2
                     else btl2 := BTLLINK btl1
    | (BTLEQR (ref (Some c1)),BTLEQR (ref (Some c2))) =>
        doC1 c1 c2; btl2 := BTLLINK btl1
    | _ => crash ()
  end

fun union n1 n2 = (* Make nodes 'equal' *)
  if n1 = n2 then ()
  else (unionL n1 n2;
   case (!n1,!n2) of
     (BTVAR (dps1,_),BTVAR (dps2,_)) =>
       (dps2 := dps1 @ dps2; (* O(1) list append *)
        bt1 := BTLINK bt2)
   | (BTVAR (dps,_),DYNAMIC _) =>
       (apply update_dps (!dps); bt1 := BTLINK bt2)
   | (DYNAMIC _,BTVAR (dps,_)) =>
       (apply update_dps (!dps); bt2 := BTLINK bt1)
   | _ => crash ())
```

Figure 66: Normalization algorithm — part two.

The total number of unify-operations performed is bounded by twice the number of variables. The number of find operations is bounded by four times the number of constraints processed.

All in all the algorithm can be made to run in $O(n \cdot \alpha(n, n))$ time, where n is the size of input, measured as the total number of components in the constraint set, and α is an inverse of Ackermann's function.

```
fun norm (EQUAL (l,r)) = union (find l,find r)
  | norm (LIFT (l,r)) = unionL (find l,find r)
  | norm (LEQ (l,r)) =
      let val c as (l,r) = (find l,find r)
          val eqr = geteqr r
          val mem = getmem eqr
      in case !mem of
           None => if isDL eqr then doC2 c else mem := Some c
         | Some c' => doC1 c c'
      end

fun eq_solve (EQUAL _) = ()
  | eq_solve (LIFT (l,r)) =
      let val (l',r') = (find l,find r)
      in if isvar r' andalso l' <> r' then r' := BTLINK l else ()
      end
  | eq_solve (LEQ c) = eq_solve (LIFT c) (* ! *)

fun eq_S n = if isvar n then n := BTLINK Snode else ()

fun generate () = ... (* Generate constraints *)

fun normalize () = (* Assumes phase 1 done during generation *)
  let val save = !worklist
  in worklist := radixsort (!worklist); (* O(n), LEQs last *)
     (* From now on LEQs are only chosen if there's no EQUAL/LIFT *)
     while !worklist <> [] do
       let val (c :: rest) = !worklist
       in worklist := rest;
          norm c
       end
     apply eq_solve save;
     apply eq_S btvarlist
  end
```

Figure 67: Normalization algorithm — part three.

Debuggability The binding-time analysis described in this chapter and others using the same principle of constraint normalization may have many appealing properties, but they all share one irritating one: they are extremely difficult to debug! First of all, even very small programs generate hundreds of variables and thousands of constraints; there is no such thing as a small example. Second, the state of the algorithm is represented by equivalence relations which in turn are represented by some kind of pointers. That makes the state very difficult to track and some intermediate state may be difficult to interpret. Third, a programming mistake need not show itself for months, but some day all of a sudden the algorithm may then just enter an infinite loop or produce the wrong result. Fourth, the data structures may be cyclic (since the binding times may be cyclic) so test-printing is difficult.

These findings have all been confirmed by Peter Holst Andersen at DIKU who has done the majority of the implementation of the C-Mix binding-time analysis described in [And92]. The reader is now warned that extreme care is needed when programming this type of binding-time analysis!

5.8 Summary and Related Work

We have developed a binding-time analysis for Standard ML via non-standard type inference and shown that it can be implemented very efficiently, both in theory and practice. We have also shown that the same principles smoothly can be used for other analyses such as the analysis that identifies which of the multi-branch lambda expressions that cannot be unfolded due to lack of static information. The achievement of our binding-time analysis is the extension to a really large language, the simplicity obtained by using the type information, the ability to handle complicated patterns, and the ability to handle exceptions.

The first work on binding-time analysis by means of type inference seems to be [Gom89] but at that time `lift`-insertion was not integrated, and the algorithm had a worst case of cubic running time (for the pure λ-calculus without let-expressions).

The major breakthrough came with [Hen91] where Henglein showed that insertion of `lifts` could be integrated into the analysis *and* that it was possible to implement efficiently. Both Gomard's and Henglein's analyses were for the untyped λ-calculus, but implicitly imposed a standard type discipline on the language by making offending constructs dynamic.

An extension of the method was used in [And92] for the C language. As the number of different "types" were increased so were the number of rewriting rules, but as the source language for this analysis is strongly typed the number of rewriting rules is still reasonable. Later in [And93] an improvement on insertion of `lifts` was presented. The basic idea is to move the lift predicates from the place a value is generated to the place it is used such as array indexing. By using the splitting of expressions into atomic and non-atomic expression we have obtained a different optimization of approximately the same size.

The basic principles of Henglein's method were used in [BJ93] in an analysis for the Scheme language. The set of binding-time values is completely different from the above analyses, so we will not go further into that method here.

Binding-time analysis based on type inference was originally proposed by the Nielsons in [NN88], but their work has until now proven unsuitable for application to partial evaluation.

Completely different methods have been used for binding-time analysis, mostly based on some form of projections, also called divisions [JGS93]. Most have been based on abstract interpretation and fixed-point iteration. In the first analyses for flowchart-mix [JGS93, Chapter 4] a two-point domain of $\{\mathtt{S},\mathtt{D}\}$ was used and that was later extended in [Bon91] where a four-point domain is used to handle also higher-order functions. A different approach was used in [Lau89] and later in [dN93] where the fixed-point iteration is over sets of domain projections, each projection mapping values to their static parts. These methods have been made to work with both polymorphism and polyvariance.

Chapter 6

Implementation and Experiments

> **No technique works if it isn't used.** *If that sounds simplistic, look at some specifics:*
>
> *Telling friends about your diet won't make you thin. Buying a diet cookbook won't either. Even reading the recipes doesn't help.*
>
> *Knowing about Alcoholics Anonymous, looking up the phone number, even jotting it on real paper, won't make you sober.*
>
> — LARRY NIVEN, *Niven's Laws, quoted from N-Space (1990)*

Based on the methods and analyses described in the previous chapters we have implemented a system for partial evaluation of Standard ML. We have not implemented everything so in this chapter we will describe what we have implemented and what we have not. After that we report on some experiments with the system.

6.1 Implementation

First of all: the implementation does not include anything about exceptions and side effects. Keep this in mind when we below write that we have implemented this or that. Second, as specified below we have used code from the ML Kit, [BRTT93]. We wish to make it very clear that the ML Kit-code should not be considered part of our work.

Preprocessing Of the preprocessing phase we have implemented a parser (by taking the parser included in the ML Kit and making some minor interface changes), a type-checker (again using the code from the ML Kit and making some minor changes in order to have the syntax tree annotated with types), alpha-conversion, and elimination of flexible records (three dots in record patterns). The binding-time analysis and program annotator have been implemented, but do not insert sp-functions at dynamic lambdas.

We have not implemented simplification of value bindings, elimination of `local`-phrases, elimination of `abstype`-declarations elimination of `op`-clauses, elimination of pattern wildcards, making matches exhaustive, and data type globalization. These are trivial to implement. Furthermore we have not implemented unfolding of polymorphism and `let`-insertion. These are not entirely trivial.

Compiler generation The compiler generator has been implemented.

Postprocessing We have implemented a pretty-printer by taking the one in the ML Kit, making some changes, and fixing some bugs.

* * *

The total size of the system is approximately 43000 lines of Standard ML code of which approximately 34000 lines are from the ML Kit. The binding-time analysis is approximately 2500 lines, the compiler generator is 3000 lines, and other parts written by us total approximately 3500 lines. The ML Kit-code used contains only few comments[1] while our code is reasonably well-documented. When the system is compiled and turned into an executable file, the file size is approximately 13 MB. The compilation will use any amount of available memory and still garbage-collect repeatedly; the compiled program does not generate large amounts of garbage, though. The program has been compiled on host Embla at DIKU which is a HP 9000 series 700. 50MB of memory has been made available to the process. The compiler used is SML of New Jersey version 0.93 (beta release). The timings reported below were made using this setup.

6.2 Experiments

We have made some experiments with our system for partial evaluation and report here on two. The first one, an interpreter, uses a lot of data types, pattern matching, but few calculations. The second one, Ackermann's function, on the other hand uses a lot of calculations. The interpreter is coded primarily with tuples, while Ackermann's function is curried.

6.2.1 Interpreter for Imperative Language

From [JGS93, figure 3.3] we have taken an interpreter for a small imperative language. It has been modified slightly to remove polymorphism in lists and expanded with a few more operators, but lets have not been inserted. This interpreter has been used with the following program for computing the greatest common divisor of two natural numbers (taken also from [JGS93]):

```
1: if x = y goto 7 else 2
2: if x < y goto 5 else 3
3: x := x - y
4: goto 1
5: y := y - x
6: goto 1
7: return x
```

which written in the abstract syntax can be seen in Figure 68. If lines 3 and 5 had used the modulo operator instead this would have been Euclid's algorithm.

[1] ...in the code, but has lots of external documentation.

```
PGMCONS(COND(OP("=",EXPCONS(VAR "x",EXPCONS(VAR "y",EXPNIL))),7,2),
PGMCONS(COND(OP("<",EXPCONS(VAR "x",EXPCONS(VAR "y",EXPNIL))),5,3),
PGMCONS(ASSIGN("x",OP("-",EXPCONS(VAR "x",EXPCONS(VAR "y",EXPNIL)))),
PGMCONS(GOTO 1,
PGMCONS(ASSIGN("y",OP("-",EXPCONS(VAR "y",EXPCONS(VAR "x",EXPNIL)))),
PGMCONS(GOTO 1,
PGMCONS(RETURN (VAR "x"),
PGMNIL)))))))
```

Figure 68: flow-chart program for greatest common divisor.

Running our compiler generator with the interpreter as input and a store specifying known variable names but unknown values should now produce a compiler from the flow-chart language to Standard ML. It does. It is 734 lines long and appears in appendix B. When used with the program example it produces the following program

```
fun f3 b = if b then 0 else 1
and f1 b = if b then 0 else 1;
fun f2 (x,y,true)  = x
  | f2 (x,y,false) = f4 (x,y,f3 (x<y) = 0)
and f4 (x,y,true)  = f2 (x,y-x,f1 (x=y-x) = 0)
  | f4 (x,y,false) = f2 (x-y,y,f1 (x-y=y) = 0);
fun main (x,y) = f2 (x,y,f1 (x=y) = 0);
```

which for the purpose of inclusion here has been alpha-converted and slightly sugared. Assuming that the ML-compiler it smart enough to convert the program to continuation passing style, to perform constant folding, and to unfold trivial functions, this program is optimal. The running times are

Program	Runs	Mean running time (ms)	Relative
Interpreter	2000	2.50	125
Compiled program	50000	0.02	1

These timings do not include garbage collection, but that amounts to less than 1% for the interpreter and is unmeasurable for the compiled program (which probably does not generate garbage at all). The relative quotient was computed from the timings *before* rounding and its accuracy is ± 1.

The factor 125 is extraordinary large (speed-ups reported in the literature for the same kind of problem has been around a factor of 5–20); it has been checked and double checked — it is correct. Optimizations on the interpreter lower the factor to approximately 85. This is still a very large factor and indicates that pattern matching is a very expensive operation. Interestingly enough, the residual program remains constant under the optimizations, so if the residual program was what we wanted, then there was no gain at all by optimizing the interpreter.

6.2.2 The Ackermann Function

Although Ackermann's function is completely useless for practical purposes it is traditionally used for evaluating partial evalutors. We have used the following curried version:

```
val rec ack = fn m => fn n =>
  (fn 0 => n + 1
   | m' => (fn 0 => ack (m' - 1) 1
            | n' => ack (m' - 1) (ack m' (n' - 1))) n) m
```

We assume that `m` is static while `n` is dynamic. The corresponding two-level program including inserted sp-functions was shown in Example 2. Note that we have made the pattern matching directly on the integer instead of using `if`-expressions.

The generating extension shown in Appendix C is 134 lines long. We used the generating extension to specialize Ackermann's function to `m` = 3. The specialized program was:

```
fun f3 0  = f2 1
     | n' = f2 (f3 (n' - 1))
and f2 0  = f1 1
     | n' = f1 (f2 (n' - 1))
and f1 0  = 1 + 1
     | n' = f1 (n' - 1) + 1
```

where some syntactic sugar (alpha-conversion, reintroduction of infix operators, and identification of $\{1 : \tau\}$ with τ) has been added. The reason that the constant expression `1+1` is left in the residual program, is that we use monovariant specialization. The expression originates from the first recursive call above, where the constant `1` is lifted.

To measure the gain from partial evaluation we used both the residual program and the original to evaluate `(ack 3 8)` (which is 2045 with the definition above). The running times were:

Program	Runs	Mean running time (s)	Relative
ack	10	4.598	6.8
ack_3	100	0.680	1

Note that the running times is given in seconds, while they in the previous table were given in milli seconds. It is expensive to calculate Ackermann's function. The speed-up here originates partly from the reduced number of pattern matchings and partly from the reduced number of function applications.

6.3 Summary

In this chapter we have described our implementation and provided benchmarks documenting the systems performance. We applied it to the Ackermann function and an interpreter, and we saw that it was able to remove a complete layer of interpretation. The benchmarks are very good.

```
datatype exp     = NUM of int                                (* Integer constant     *)
                 | OP of string * explist                    (* Operator application *)
                 | VAR of string                             (* Variable             *)

     and explist = EXPCONS of exp * explist                  (* List of expressions  *)
                 | EXPNIL

datatype command = ASSIGN of string * exp                    (* Assignment command   *)
                 | COND of exp * int * int                   (* If-then-else command *)
                 | GOTO of int                               (* Goto command         *)
                 | RETURN of exp                             (* Return command       *)

datatype pgm     = PGMCONS of command * pgm                  (* List of commands     *)
                 | PGMNIL

datatype store   = STORECONS of string * int * store         (* List of variables    *)
                 | STORENIL

fun nth_command (1, PGMCONS(c,pgm)) = c                      (* Find nth command *)
  | nth_command (n, PGMCONS(c,pgm)) = nth_command(n-1, pgm)
  | nth_command (n, PGMNIL)         = GOTO (~1)              (* Dummy *)

fun lookup (x, STORENIL)              = ~1                   (* Find variable *)
  | lookup (x, STORECONS(x',v',store)) = if x = x' then v' else lookup(x,store)

fun update (STORENIL, x, v) =  STORECONS(x,v,STORENIL)       (* Update variable *)
  | update (STORECONS(x',v',s),x,v) =
      if x = x' then STORECONS(x,v,s)
      else
        let val s' = update(s,x,v)
        in STORECONS(x',v',s')
        end

fun eval (NUM n, s)                                   = n   (* Evaluate expression *)
  | eval (VAR x, s)                                   = lookup (x, s)
  | eval (OP("+",EXPCONS(e1,EXPCONS(e2,EXPNIL))), s) = (eval(e1, s))+(eval(e2, s))
  | eval (OP("-",EXPCONS(e1,EXPCONS(e2,EXPNIL))), s) = (eval(e1, s))-(eval(e2, s))
  | eval (OP("=",EXPCONS(e1,EXPCONS(e2,EXPNIL))), s) =
    if (eval (e1, s)) = (eval (e2, s)) then 0 else 1
  | eval (OP("<",EXPCONS(e1,EXPCONS(e2,EXPNIL))), s) =
      if (eval (e1, s)) < (eval (e2, s)) then 0 else 1
  | eval _                                            = ~1  (* dummy *)

fun run (l, GOTO n, s, p)        = run (n, nth_command (n, p), s, p)
  | run (l, ASSIGN(x, e), s, p)  = let val s1 = update (s, x, eval(e, s))
                                   in
                                     run (l+1, nth_command(l+1, p), s1, p)
                                   end
  | run (l, COND(e,m,n), s, p)   = if eval (e, s) = 0
                                   then run (m, nth_command(m, p), s, p)
                                   else run (n, nth_command(n, p), s, p)
  | run (l, RETURN e, s, p)      = eval (e, s)

fun interpret (PGMCONS(c,pgm)) s = run (1, c, s, PGMCONS(c,pgm))
  | interpret PGMNIL _           = ~1                         (* dummy *)
```

Figure 69: Interpreter for flow-chart language.

Chapter 7

Conclusion

Did this Caesar seem ambitious? When that the poor have cried, Caesar hath wept; ambition should be made of sterner stuff: yet Brutus says he was ambitious, and Brutus is an honourable man.

— WILLIAM SHAKESPEARE, *Julius Caesar, III.2 (1599)*

7.1 Contributions of this Work

The work reported in this thesis has contributed to the field of partial evaluation on a number of points:

- This is the first full-scale attack on a large, typed, functional language. Previous partial evaluators have treated significantly smaller languages, at least during the actual partial evaluation process.
- The coding and untagging problems have been completely worked around.
- It has proven possible to perform partial evaluation by writing a compiler generator directly instead of writing a traditional partial evaluator. This has to some extent been done before but only for untyped toy languages. We believe that eventually this approach to partial evaluation will dominate because it has proven superior.
- The binding-time analysis presented here is remarkably simple even though it handles a larger and richer language than ever before. The typedness of the source language has been used to avoid effectively redoing the standard type inference and checking in the binding-time analysis.

  We have chosen to treat the large source language without translating it into a minimal language. We strongly believe that if such a translation is performed, then it should be performed *after* the binding-time analysis, so it does not make the partial evaluation degenerate by arbitrary choices of, for example, pattern matching order.

- Partial evaluation of complicated patterns has succeeded for the first time. Previous attempts have failed to handle patterns as complicated as Standard ML's, i.e., with matching on multiple constants/constructors simultaneously and variable bindings.
- Partial evaluation of local recursive functions has succeeded for the first time. Other partial evaluator systems that have been able to treat local recursive functions, have only been able to do so by lambda-lifting. See the sections on Similix and Schism below. The impact of using our approach is not known at this time.
- Partial evaluation of exceptions has succeeded for the first time, but has not yet been implemented. This must be seen in the light that it is also the first time it has been attempted. We believe however, that treating exceptions reasonably must be done by compiler generation; a normal partial evaluator would have to tag all values explicitly which is not reasonable.

7.2 Related Work

In this section we will sketch the connections between the work presented in this thesis and the work of others in the same fields. We start out with different other systems for partial evaluation and then turn to work that is not directly connected to such an implemented system. The latter category also includes work in the traditional field of compiler generation.

7.2.1 Similix

The Similix system for partially evaluating programs written in a subset of Scheme (an untyped dialect of Lisp) is probably the best established partial evaluator available today, and we have used it as a reference system. It was originally written by Anders Bondorf and Olivier Danvy and has been described in [BD91], [Bon90], [Bon91], [BJ93], [JGS93] and several other places. Similix is an off-line, self-applicable partial evaluator that handles higher-order functions. It can, unlike our system, specialize with respect to higher order values; to determine if a higher-order value has been seen before, Similix compares code for the functions and the values of free variables. Similix supports partially static structures for user defined types (even though Scheme is a dynamically typed language) and can handle side effects to a certain degree; all side effects are put into the residual program. Just like our system, Similix is capable of handling recursive let-bindings, but it is done by lambda-lifting. The specializer cannot handle recursive let-bindings.

Earlier versions of Similix, like the one described in [Bon91], used abstract interpretation and fixed-point iteration over a four-point domain (with values $\perp$, `S`, `Cl`, and `D`) to perform the binding-time analysis, while the current version of Similix system uses a constraint based binding-time analysis. While the former analysis was stronger than the current (see [PS91]) it was a cubic-time algorithm (worst case) and the loss in strength was small enough to be accepted in return for very significant speed-up. Aside from using constraints and union-find techniques, the binding-time analysis is radically different from the one presented in this thesis. All function information is analyzed in a separate pass, the flow analysis, and functions are thereafter just represented as closures of some arity. The flow-analysis performs a function similar to what we do with our variable equivalence relation. We strongly believe that the flow analysis and the binding-time analysis developed in [BJ93] could be merged into one using our principle of a growing variable equivalence relation.

Bondorf and Jørgensen have for the first time implemented the termination improvement suggested in [Hol88] and it is reported to prevent infinite specialization in many typical situations.

7.2.2 C-Mix

In his master's thesis, [And92], Lars Ole Andersen describes the development of an off-line self-applicable partial evaluator for a subset of the C language. A revised version of the essential parts can be found in [JGS93]. Syntactically most of the C language is covered but the only base type allowed is `int` and pointers may point to arrays only. Arrays must be declared statically, but in later work [And93] dynamic allocation has been allowed.

The binding-time analysis used is similar to the one presented in this thesis, but it uses separate binding-time values for structures (essentially product types) and pointers, even though the language is strongly typed. Another difference is that the lift predicates are generated where values are needed, say when indexing an array, and not where they are generated. This reduces the number of binding-time variables needed.

7.2.3 Petrarca

The Petrarca system developed by Anne de Niel in her Ph.D. thesis [dN93] is an off-line partial evaluator for a small strongly typed first-order language, Potyful. Potyful is polymorphic and allows the user to define her/his own data types. Besides type information, a program consists of a series of function definitions. Pattern matching is available in the language but only in a simple version allowing identification and removal of tags. In other words, all patterns have the form `CONSTRUCTOR var` so they are not nearly as complicated and powerful as patterns in Standard ML.

The binding-time analysis is both polymorphic and polyvariant, and uses projections from domain theory. It is done by abstract interpretation and fixed-point iteration and it results in a set of binding-time projections for each function. It turns out, however, that this set is too large as *de facto* equivalent functions are output. de Niel states this as the fact that prohibits self-application of the specializer. No estimate of the complexity of the algorithm is given but it cannot be better than cubic. The binding-time analysis seems to be an enhancement of the one developed in [Lau89].

The Petrarca system uses program bifurcation, a process that splits partially static structure variables into several either static or dynamic variables, so the specializer itself gets simpler. A drawback of this transformation is that many unused variables are created. To reduce the number of residual functions a useless-variable analysis is developed to remove the unused ones so functions do not get specialized with respect to those. This problem and its solution is a variant of the early work with liveness of static variables described for instance in [JGS93, Section 4.9.3].

7.2.4 λ-mix

A self-applicable off-line partial evaluator for the untyped λ-calculus with constants and an explicit fixed-point operator was the first partial evaluator to be proven correct, in Gomard's master's thesis [Gom89] and in [Gom92]. It should be noted that the termination part of the proof requires that all subterms of the λ-term in question terminate for all environments, i.e., that they are total.

Gomard first used type inference similar to the one presented in this thesis, but using algorithm W, to do the binding-time analysis. The method for automatic lift insertion was later developed by Henglein [Hen91] and used in the presentation of λ-mix in [JGS93].

7.2.5 Schism

The Schism system [Con93] is an off-line partial evaluator for a subset of Scheme (like Similix), and a front-end for a subset of Standard ML has been added. As the language that Schism does partial evaluation on is very simple, many aspects of the handling of Standard ML are hidden well in the translator. For example: Schism lets the user define sum and product types and can handle pattern matching, but the matching is not as general as pattern matching in Standard ML: it is limited to tag identification and removal plus tuple splitting. In other words patterns must have the form `CONSTRUCTOR (v1,...,vn)` or `CONSTRUCTOR v`. This means that the front-end translator from Standard ML to Scheme must include some kind of a pattern matching compiler choosing some kind of matching sequence for complicated patterns. As this happens *before* the binding-time analysis it may lead to unwanted results. The other main hidden thing is recursive let-binding. These must somehow be globalized because the only recursive construct available is global, and that means abstraction over free variables. The impact of this translation is unknown, but will probably depend on the ML system used to run the residual program and its efficiency in building closures.

The subset of Standard ML treated by the translator is basically the Core of Standard ML [MTH90] but without imperative features (references, input/output, and exceptions) and records.[1]

The binding-time analysis used is polyvariant and based on fixed-point iteration, but little more is known at this time, as it is not yet published.

[1] It is clearly stated in [Con93] that the translator does not handle records. However, in the same article Consel presents a translation example that includes records in the form of tuples. We therefore strongly suspect that it would be extremely simple to expand Schism to records by using type information to convert records to tuples.

7.2.6 Text Books

In [Pag91] Frank D. Pagan discusses partial evaluation of Pascal by hand-writing generating extensions after what corresponds to a manual binding-time analysis. The fundamental principles in Pagan's generating extensions are similar to the ones our compiler generator produces: some calculations are performed by the generating extension, the rest are output to the residual program. One important difference is that Pagan's extensions do not memorize. To prevent infinite unfolding of `while`-statements with static condition he uses human insight about a controlling variable's bounded number of possible values and uses "the trick"[2] [JGS93] for that variable. The net effect of this is the same as obtained in [And92] by rewriting `while`-statements into `goto`-statements. Pagan uses his methodology on a number of examples including a general table-driven LR(1)-parser like the ones produced by the standard Unix tools YACC and Gnu Bison. A speed-up of approximately two is reported for these already optimized compiler generator tools! Pagan also demonstrates how the methodology can be used to derive compilers from some language to the host language (Pascal).

In [JGS93] the authors cover the basics of partial evaluation of a number of languages spanning from C to Prolog and the λ-calculus. The book is basically compiled from the authors's and invited writers's articles and covers most known techniques and results on off-line partial evaluation. The main reason for mentioning the book here, where we also cover most of the articles it is based on separately, is that it has an extensive guide to the literature and an overwhelming bibliography.

Another approach to compiler generation is taken in [Lee89]. Lee uses the essence of the first Futamura projection (equation 1.1) that an interpreter (denotational semantics) given the program but not the runtime data can be reduced to a compiled version of the program, but the reduction types considered are only α-, β-, and η-reductions of the interpreter. Lee, referring to [Mos82], observes that the low-level structure of the interpreter such as the use of closures to model stores will often be left in the compiled program. Lee therefore creates a two-level way of specifying semantics: the micro-semantics and the macro-semantics. In other words he effectively writes a two-pass interpreter which is a direct analog to specifying the semantics of a two-level language except that the static and dynamic part of the languages need not be the same. The "micro" part of the semantics corresponds[3] to the residual part of the semantics of our two-level language, so the language of the micro-semantics is the intermediate output language of the compiler. On top of the compiler Lee adds a code generator (which might be seen as a compiler from the micro language to the final output language; this is assembler for the Intel 8086 processor). At runtime the generated compilers outperform commercial compilers like Turbo Pascal (probably version 3) from Borland International, even though that compiler notoriously generates slow code in a hurry [Wel90].

[2] Dynamic entities ranging over a finite set of possible values can be made static by explicitly comparing it with all those values, and then continue evaluation with the constant values.

[3] More precisely: it should correspond to the residual part of the semantics of our two-level language, if the separation into "micro" and "macro" is performed correctly.

7.2.7 Other Work

In [BHOS76] the authors report on a so-called partial evaluation compiler, *RedCompile*, which is a hand-written compiler generator for Lisp. This article seems to be the origin of the idea of hand-writing cogen instead of mix, but the goals are different: Beckman *et al.* use the idea to gain efficiency as cogen is to compilers what mix is to interpreters. *RedCompile* relies on user-annotation of functions to ensure preservation of semantics.

In his Ph.D. thesis, [Lau89], John Launchbury describes partial evaluation of a small typed first-order language, PEL, with user defined data types. Two versions of the language are considered: with and without polymorphism. Launchbury develops a binding-time analysis based on projections from domain theory and succeeds in making that analysis for the polymorphic language. PEL has pattern matching but the pattern matching construct is limited to tag identification and removal, i.e., only patterns of the form `CONSTRUCTOR var` are treated. It remains an open question whether the projection method can be expanded to handle more powerful pattern matching as in Standard ML. As the partial evaluator for PEL itself is written in a completely different language, it is not self-applicable.

In [Mog93] Torben Mogensen considers specialization of *constructors* with respect to static parameters, equivalent to the usual specialization of *functions* with respect to static parameters. This means that dynamic constructors no longer are required to have dynamic parameters, and more static information can be used. In the article Mogensen demonstrates that partial evaluation of general parsers becomes feasible without having to rewrite the general parser using continuation passing style as in [Mos93a]. Specialization involving constructor specialization seemingly must be done by some kind of fixed-point iteration, as case-constructs must be reconsidered whenever a new specialized constructor is generated. Until another way of specializing is found, there seems to be no hope of performing constructor specialization using the compiler generation approach — at least not with the control structure we use.

In [Lau91] Launchbury succeeds in writing the first self-applicable partial evaluator for a strongly typed language, namely a subset of Lazy ML. The paper focuses on the problems of writing a self-applicative partial evaluators for a typed language, such as the problem of double-encoding; these problems have been avoided by the methods used in this thesis. The self-application is reported as successful, but the Ackermann example in the article shows that there's still a lot of specializer parts left in the compiler — the result is not nearly as nice as our result. Left-over is also a lot of value-tagging and untagging, see [And92].

In [Hol89] Carsten Kehler Holst describes what he calls "Syntactic Currying." It is basically the same as we call partial evaluation by compiler generation. The language used is a minimal subset of Scheme without higher-order functions and side effects. The system is implemented in Scheme using macros. Due to the choice of language, Holst did not realize the real value of his findings when used on a typed language; he and Launchbury did that later in [HL92].

Partial evaluation of Standard ML was recently attempted in [Blo93]. This master's thesis is in Dutch[4] so we do not understand everything in it. It is however clear that it treats a significantly smaller language that we do; it is a first-order subset and exceptions are not handled. The actual partial evaluation is done by using the Petrarca system (see above) and large parts of the thesis is on parsing and type-checking of Standard ML.

In his just-finished master's thesis [Mos93b] considers polyvariant binding-time analysis. Normal (unconstrained) polyvariant binding-time analysis tend to produce an unreasonable number of variants, so Mossin describes the effect of using a polymorphic type-system for the binding-time types.

7.3 Future Work

On the basis of our work, we find that a number of white spots on the map should be investigated:

- A lot of holes in the implementation should be filled-in. This will enable us to gather more experimental results to evaluate the cogen approach and our fundamental design decisions.
- Continuation based specialization should be implemented. As the situation is now, the `let`-insertion will ruin the binding times too often unless the source program itself is written in continuation passing style.
- On the theoretical side it should be considered whether constructor specialization and the cogen approach go together well.

7.4 Conclusion

Rereading the thesis stated in the preface we conclude: 1) Yes, it did indeed prove possible in practice to write compiler generator for partial evaluation by hand. 2) Yes, things were simpler in the cogen-setting, as the coding problem did disappear. 3) Yes, it was most certainly correct that certain hard problems in traditional partial evaluation became much simpler in the cogen-setting; this turned out to be true not only for patterns but also for exceptions. All in all we feel that our thesis has been proved completely true by our work.

[4] Dutch is close enough to Danish and German to allow reading of some of the text.

Kapitel 8

Dansk resumé

Hvor smiler fager den danske kyst
og breder favnen, når solklar bølge
og sommerskyer og skib med lyst
står sundet ind i hinandens følge,
og Kronborg luder
ved Sjællands port
mod hvide skuder,
hvor lyst! hvor stort!

— JOHANNES V. JENSEN, *Hvor smiler fager*

Det er i bund og grund en falliterklæring ikke at kunne skrive sit speciale på sit modersmål, men som verden er skruet sammen i dag, ville specialet med sikkerhed ikke blive læst uden for landets grænser, hvis det havde været på dansk. Af hensyn til danske læsere giver vi i dette kapitel et dansk resumé af specialet.

Vort arbejde med specialet har haft sin baggrund i følgende, tredelte tese om partiel evaluering.

> *Partiel evaluering med en håndskreven oversættergenerator er 1) praktisk muligt, 2) enklere end traditionel partiel evaluering og 3) bedre egnet til at angribe nogle af de svære problemer inden for partiel evaluering, såsom mønstersammenligning.*

Vi præsenterer i kapitel 1 de to forskellige tilgange til partiel evaluering: den traditionelle fortolkerlignende tilgang, der karakteriseres ved den såkaldte mix-ligning (side 10), og "vores" oversætterlignende tilgang, hvis virkemåde nok bedst beskrives af ligningerne 1.4–1.6. Uformelt kan man sige, at partiel evaluering ved hjælp af en oversættergenerator sker ved, at et programs beregninger deles op til to grupper, alt efter om de udelukkende afhænger af statiske værdier (såkaldte statiske beregninger), eller om de også afhænger af dynamiske værdier (såkaldte residuelle beregninger). Oversættergeneratorens opgave er så at producere et program, der udfører de statiske beregninger, medens det laver kode til de dynamiske.

Alt dette lyder rimeligvis som en ganske indviklet måde at foretage partiel evaluering på, så der må være nogle fordele til at forsvare tilgangen — fordelene diskuterer vi i kapitel 1. Først og fremmest er der ingen sprogbegrænsninger ved programmeringen af oversættergeneratoren, fordi det ikke er et mål at kunne selvanvende den. For det andet slipper man for at skrive en fortolker for det sprog, som man laver en partiel evaluator for. En sådan fortolker har man brug for ved traditionel partiel evaluering, idet alle statiske beregninger skal foretages af mix. For det tredje løber man ikke ind i det tekniske, men alvorlige dobbeltkodningsproblem, der kan få programmer til at svulme op til groteske størrelser.

Vores oversættergenerator er bygget til at håndtere sproget Standard ML, der er defineret i [MTH90] og kommenteret i [MT91]. Vi har ikke bekymret os om modulsproget, men har koncentreret os om kernesproget. Vi beskriver i kapitel 2 tre restriktioner, vi har været nødt til at indføre for programmer. De to er banale og uinteressante, men den sidste er ikke: vi kan ikke håndtere polymorfi og må derfor kræve, at programmets hovedfunktion ikke er polymorf. Programmet må gerne benytte polymorfi intern. I kapitlet beskriver vi desuden, hvorledes man kan foretage en række forenklinger af SML-programmer på en sådan måde, at de bliver lettere at håndtere, men uden at resultatet af den partielle evaluering bliver forringet. Det betyder for eksempel, at vi bevarer de komplicerede mønstre i stedet for at "udflade" dem, som for eksempel [Con93] har gjort, for en sådan udfladning kan ødelægge et programs bindingstidsseparation på uforudsigelig måde. Vi kalder det resulterende sprog for det simplificerede kernesprog.

Vi går derefter i kapitel 3 over til at diskutere en lang række forskellige aspekter af partial evaluering for programmer skrevet i det simplificerede kernesprog. Vort mål har været at kunne håndtere et så stort sprog som muligt med kalkuleret risiko for, at det kunne gå ud over styrken af den partielle evaluator. Det *er* gået ud over styrken, idet vi for eksempel kun specialiserer med hensyn til værdier, som kan sammenlignes med lighedsoperatoren. Hvis vi også skulle specialisere med hensyn til funktioner, ville vi løbe ind i alligevel at skulle kode værdier. Konsekvenserne af disse restriktioner kendes endnu ikke, men vi tror, at de ikke er så alvorlige i praksis.

Vi har valgt at benytte en monovariant bindingstidsanalyse, idet en sådan kan implementeres særdeles effektivt, og fordi man stadig ikke har kunnet styre graden af polyvarians fornuftigt. For at undgå dyre og overflødige funktionskald i residualprogrammet, benytter vi udfoldningsstrategien fra Similix [JGS93]. Den går i korthed ud på at udfolde samtlige funktionskald i kildeprogrammet samt at indsætte specialiceringspunkter ved dynamiske lambdaudtryk og dynamiske betingelser. Strategien giver gode resultater, men kan i visse konstruerede tilfælde føre til uendelige udfoldning, altså ikke-termination af specialiserings processen.

For at spore os ind på, hvorledes en oversættergenerator skal konstrueres, går vi derefter over til at undersøge, hvilken kontrolstruktur de genererede oversættere skal have. Det sker i afsnit 3.2. Vi præsenterer to vidt forskellige alternativer, ét lånt fra [HL92] og ét, vi selv har designet. Førstnævnte er bredde-først-baseret og er rent funktionelt, medens sidstnævnte er dybde-først og kræver muligheden for at kunne foretage destruktiv opdatering af variable. Vi demonstrerer, at vores løsning er overlegen til den givne opgave, både hvad angår effektivitet og evnen til at håndtere andet end rene funktionsudtryk.

Rekursive lokale funktioner har hidtil ikke kunnet håndteres af partielle evaluatorer. Ganske vist kan de håndteres med den allersidste version af Similix systemet, men det sker ved lamdbaløftning med de sommetider uheldige konsekvenser, det kan have på effektiviteten. På grund af den kontrolstruktur, vi har valgt, samt det, at vi ikke specialiserer med hensyn til funktioner, kan vi håndtere disse lokale funktioner. Vi gør dette ved at efterlade de specialiserede udgaver af en funktion i samme omgivelser, hvori den oprindelige optrådte i kildeprogrammet. Denne fremgangsmåde kan føre til flere ens residualfunktioner placeret i forskellige omgivelser; det er uvist, hvor stort et problem dette er i praksis, hvis det overhovedet *er* et problem.

Vi fortsætter med at analysere, hvorledes partielt statiske strukturer kan behandles, altså strukturer, hvor visse dele er statiske, medens andre er dynamiske. Dette medfører i mange tilfælde, at en funktion, der i kildeprogrammet har én parameter, kan have residua med et vilkårligt antal parametre. Fænomenet kunne for eksempel optræde, hvis der var tale om en liste med kendt længde, men ukendt indhold. Resultatet kan være imponerende, som det kan ses i kapitel 6, hvor en fortolker for et lille sprog specialiseres med hensyn til et program. Resultatet er en nærmest perfekt oversættelse af programmet til Standard ML.

Mønstre er en særdeles central del af Standard ML. De er særdeles kraftfulde og ingen tidligere partiel evaluator har kunnet klare dem. Det skyldes imidlertid i høj grad det ovenfor nævnte faktum, at traditionelle partielle evaluatorer indeholder en selvfortolker. Da vores oversættergeneratorbaserede system ikke indeholder en sådan, har vi helt kunne løse problemerne[1] med mønstre. Det har tilmed været muligt at få dette til at arbejde sammen med de partielt statiske strukturer, igen med imponerende resultater til følge.

Beregningsundtagelser som de findes i Standard ML har heller ikke kunnet behandles før. I afsnit 3.9 løser vi stort set alle problemer med partiel evaluering og undtagelser, dog ikke for lokalt erklærede undtagelsestyper, som vi er tvunget til at globalisere. Atter viser det sig, at vor oversættergeneratorfremgangsmåde og kontrolstruktur er nøglen til løsningen.

Kapitlet om specialisering efterlader en række spørgsmål om sammenpasningen af de forskellige løsninger. For at besvare disse spørgsmål og for at lægge en solid grund under vores metode udvikler vi i kapitel 4 en toniveauudgave af det simplificerede kernesprog. Et program oversættes fra etniveausproget til toniveausproget ved tilføjelse af annotationer, og vi opstiller inferensregler til afgørelse af, om oversættelsen er konsistent. For dette toniveausprog viser vi produktion for produktion, præcis hvorledes kodegenereringen skal foretages. Genereringen er således syntaksdirigeret og er derfor særdeles hurtig.

Reglerne i kapitel 4 giver som nævnt mulighed for at kontrollere en oversættelse fra etniveausproget til toniveausproget, men de siger intet om, hvorledes oversættelsen skal ske. Det giver vi svaret på i kapitel 5, hvor vi udvikler en overordentlig effektiv metode til bindingstidsanalyse og programannotation. På trods af de mange klasser af værdier i Standard ML er det lykkedes os at gøre analysen enklere end tidligere analyser, og vi har for første gang demonstreret, at veltypethed af kildeprogrammet kan udnyttes

[1] Dog skal det retfærdigvis nævnes, at Standard ML's facilitet med lagdelte mønstre ikke kan håndteres med noget godt resultat. Det skyldes, at muligheden for at referere til samme værdi under to navne uvægerligt fører til kodeduplikation.

positivt i bindingstidsanalysen. Analysen bygger på løsning af et stort system af "svage uligheder," der omskrives, indtil systemet lader sig løse ved lighed. Undervejs bemærker vi, at tidligere artikler om emnet har haft visse fejl, som vi har rettet.

I kapitel 6 beskriver vi det system til oversættergenerering, som vi har udviklet og implementeret. Implementationen omfatter ikke alle beskrevne emner; især sprogsimplifikationen har nogle huller og udfoldningen af polymorfi er for eksempel ikke implementeret. Kernen er dog næsten fuldstændig, og det har været muligt at afvikle testkørsler. Som kort nævnt ovenfor fik vi ud fra en fortolker [JGS93, figur 3.3] for et lille imperativt sprog konstrueret en oversætter for samme sprog. Vi anvendte denne fortolker på et lille testprogram og opnåede en næsten perfekt oversættelse. Det oversatte program var ikke mindre end 125 gange hurtigere at afvikle end originalprogrammet under fortolkeren!

I kapitel 7 drager vi konklusioner og sammenligner med tidligere arbejder. Vi vil her blot trække frem, at vi i dette speciale har vist nye veje til løsningen af mange problemer inden for partiel evaluering. Det gælder blandt andet teknikker til indviklede mønstre, undtagelser, lokale rekursive funktioner og sprogstørrelse.

Appendix A

Support File for Compilers

This file will be included by all compilers generated by the SML compiler generator. It defines a number of support functions.

```
local
  open Cogen Cogen.OneDecGrammar
in
(* ---------------------------------------------------------------------- *)
exception ThisCannotHappen of string

(* ---------------------------------------------------------------------- *)
(* emit: string -> onedec ref -> oneexp -> unit                           *)
(*                                                                        *)
(* Add a valbind to the given code accumulator.                           *)
(* ---------------------------------------------------------------------- *)
fun emit (resname: string) (code: onedec ref) (body: oneexp) : unit =
  let
    val vb =
      case !code of
        EMPTY1dec => None
      | VAL1dec(REC1valbind vb) => Some vb
      | VAL1dec vb => Some vb
      | _ => raise ThisCannotHappen
  in
    case body of
      FN1exp _ => code := VAL1dec (REC1valbind
                     (PLAIN1valbind
                      (ATPAT1pat (VAR1atpat resname),
                       body,
                       vb)))
    | _ => code := VAL1dec
                     (PLAIN1valbind
                      (ATPAT1pat (VAR1atpat resname),
                       body,
                       vb))
  end

(* ---------------------------------------------------------------------- *)
(* getcode: unit -> onedec                                                *)
(*                                                                        *)
(* Return a dec with all predefined functions in the residual program,   *)
(* for our purposes simply return the empty dec.                          *)
(* ---------------------------------------------------------------------- *)
```

```
fun getcode ({} : unit) : onedec =
  EMPTY1dec

(* ---------------------------------------------------------------------- *)
(* new_code: unit -> onedec ref                                           *)
(*                                                                        *)
(* Return a fresh code accumulator for use with emit.                     *)
(* ---------------------------------------------------------------------- *)
fun new_code ({} : unit) : onedec ref =
  ref EMPTY1dec

(* ---------------------------------------------------------------------- *)
(* add_to_code: onedec ref -> onedec -> unit                              *)
(*                                                                        *)
(* add the onedec to code accumulator                                     *)
(* ---------------------------------------------------------------------- *)
fun add_to_code code onedec =
  code := SEQ1dec(!code,onedec)

(* ---------------------------------------------------------------------- *)
(* new_name: unit -> string                                               *)
(*                                                                        *)
(* Return a fresh symbol name for use with emit and applications.         *)
(* ---------------------------------------------------------------------- *)
val new_name_counter = ref 0
fun new_name ({} : unit) : string =
  (new_name_counter := (!new_name_counter) + 1;
   "f" ^ Int.string (!new_name_counter))

(* ---------------------------------------------------------------------- *)
(* seenB4: ('a, string) list ref -> 'a -> (bool, string)                  *)
(*                                                                        *)
(* Returns (true,name)  if (stat,name) is found in the association list.  *)
(*         (false,name) is not; the assoc list is updated with the new    *)
(*                      (value,name) pair. name is a fresh symbol.        *)
(* ---------------------------------------------------------------------- *)
fun seenB4 (seenB4list: (''a * string) list ref) (stats: ''a):(bool * string) =
  (true, #2 (List.first (fn (stats', _) => stats = stats') (!seenB4list)))
  handle
    List.First _ =>
      let val n = new_name()
      in
        seenB4list := (stats, n) :: (!seenB4list);
        (false, n)
      end

(* ---------------------------------------------------------------------- *)
(* lift_int: int -> oneatexp                                              *)
(* lift_string: string -> oneatexp                                        *)
(* lift_real: real -> oneatexp                                            *)
(*                                                                        *)
(* Return an atexp for the given constant.                                *)
(* ---------------------------------------------------------------------- *)
fun lift_int c    = ATEXP1exp (SCON1atexp (SCon.INTEGER c))
fun lift_string c = ATEXP1exp (SCON1atexp (SCon.STRING c))
fun lift_real c   = ATEXP1exp (SCON1atexp (SCon.REAL c))

(* ---------------------------------------------------------------------- *)
fun mk_op name x =  APP1exp(ATEXP1exp(VAR1atexp(name)),x)
```

```
(* ------------------------------------------------------------------- *)
fun mk_var n = ATEXP1exp(VAR1atexp n)

(* ------------------------------------------------------------------- *)
val vars2exp = map (ATEXP1exp o VAR1atexp)

(* ------------------------------------------------------------------- *)
val vars2pat = map (ATPAT1pat o VAR1atpat)

(* ------------------------------------------------------------------- *)
fun print_patlist name list =
  let
    fun print1 (ATPAT1pat(VAR1atpat v)) = output(std_out, v ^ " ")
      | print1 _ = output(std_out, "? ")
  in
    output(std_out,name ^ ": ");
    map print1 list;
    output(std_out,"\n")
  end

(* ------------------------------------------------------------------- *)
fun include_mk_sp_res_pat abst_formalsres formalsres_p replacelist =
  let
    fun replace (var as (ATPAT1pat(VAR1atpat v))) =
      let
        fun f [] = []
          | f (x::xs) = if x = "_" then [] else x::(f xs)
        fun eq (v,v') = (v' = implode (rev (f (rev (explode v)))))
      in
        ((#1) (List.first (fn (_,v') => eq(v,v')) replacelist)) handle
          List.First _ => var
      end
      | replace _ = raise (ThisCannotHappen "replace -- unknown pattern")
    fun folder p (n,pro) =
      (n-1, Some(PATROW1(Lab.mk_IntegerLab n,p,pro)))
    val patlist =
      map replace (formalsres_p @ (vars2pat abst_formalsres))
  in
    ATPAT1pat(RECORD1atpat
              (#2 (List.foldR folder (List.size patlist,None) (rev patlist))))
  end
(* ------------------------------------------------------------------- *)
fun include_sp_res_exp actualsres abst_actualsres =
  let
    fun folder p (n,pro) =
      (n-1, Some(EXPROW1(Lab.mk_IntegerLab n,p,pro)))
    val explist = actualsres @ abst_actualsres
  in
    ATEXP1exp(RECORD1atexp
              (#2 (List.foldR folder (List.size explist,None) (rev explist))))
  end
(* ------------------------------------------------------------------- *)
fun mk_sp_res_fn mrules =
  let
    val matchopt =
      List.foldR (fn mruleopt => fn match =>
                   case mruleopt of
                     None => match
```

```
                 | Some mrule => Some (MATCH1(mrule,match)))
                None
                mrules
  in
    case matchopt of
      None => raise (ThisCannotHappen "mk_sp_res_fn")
    | Some match => FN1exp match
  end


(* ---------------------------------------------------------------------- *)
val concat = List.foldR (General.curry (op @)) []

(* ---------------------------------------------------------------------- *)
fun fresh_var no : string =
   ("v" ^ no)
local
  val fresh_var_counter = ref 0
in
  fun fresh_var' v no : string =
   (fresh_var_counter :=  !fresh_var_counter + 1;
    ("v" ^ (makestring (!fresh_var_counter)) ^ "_" ^ no))
end
(* ---------------------------------------------------------------------- *)
val blankcode = ATEXP1exp(SCON1atexp(SCon.INTEGER 0))

(* ---------------------------------------------------------------------- *)
end
```

Appendix B

Flow-Chart Compiler

This is the compiler generated from the flow-chart interpreter. The support functions of appendix A are used.

```
(* use "../testprogs/tflowint.sml.gen"; *)
val _ = use "/home/hugin/birkedal/project/smlmix/sml/testprogs/include.sml";
local
  nonfix + - * / < > <= >= = <> :: ^ @
  open Cogen open OneDecGrammar
in

val code = new_code {}; val getcode = fn {} => SEQ1dec (getcode {}, ! code);
datatype
   exp_0 = NUM_0 of int | OP_0 of (string*explist_0) | VAR_0 of string
   and
   explist_0 = EXPCONS_0 of (exp_0*explist_0) | EXPNIL_0;
datatype
   command_0
    =
   ASSIGN_0 of (string*exp_0)
   |
   COND_0 of (exp_0*int*int)
   |
   GOTO_0 of int
   |
   RETURN_0 of exp_0;
datatype pgm_0 = PGMCONS_0 of (command_0*pgm_0) | PGMNIL_0;
datatype store_0 = STORECONS_0 of (string*oneexp*store_0) | STORENIL_0;
val rec sp_do_148 =
        fn p : pgm_0 =>
           fn no =>
              (fn PGMCONS_0 x =>
                  let val (actualsres_1,gensplitexp_1, genseenB4_1, newvars_1)=
                          sp_do_149 x (^ (no, "1"))
                  in  (actualsres_1,
                       PGMCONS_0 gensplitexp_1,
                       PGMCONS_0 genseenB4_1,
                       newvars_1
                      )
                  end
                  |
                  PGMNIL_0 => (nil, PGMNIL_0, PGMNIL_0, nil)
              )
```

```
          p
and sp_do_149 =
    fn p : {2 : pgm_0, 1 : command_0} =>
       fn no =>
          let val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_148 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
                  ) =
                  sp_do_150 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
          in  ((@ (actualsres_2, actualsres_1)),
               (gensplitexp_1, gensplitexp_2),
               (genseenB4_1, genseenB4_2),
               (@ (newvars_2, newvars_1))
              )
          end
and sp_do_150 =
    fn p : command_0 =>
       fn no =>
          (fn ASSIGN_0 x =>
              let val (actualsres_1,
                        gensplitexp_1,
                        genseenB4_1,
                        newvars_1
                       ) =
                       sp_do_156 x (^ (no, "1"))
              in  (actualsres_1,
                    ASSIGN_0 gensplitexp_1,
                    ASSIGN_0 genseenB4_1,
                    newvars_1
                  )
              end
              |
              COND_0 x =>
              let val (actualsres_2,
                        gensplitexp_2,
                        genseenB4_2,
                        newvars_2
                       ) =
                       sp_do_155 x (^ (no, "2"))
              in  (actualsres_2,
                    COND_0 gensplitexp_2,
                    COND_0 genseenB4_2,
                    newvars_2
                  )
              end
              |
              GOTO_0 x =>
              let val (actualsres_3,
                        gensplitexp_3,
                        genseenB4_3,
                        newvars_3
                       ) =
                       sp_do_143 x (^ (no, "3"))
              in  (actualsres_3,
                    GOTO_0 gensplitexp_3,
                    GOTO_0 genseenB4_3,
                    newvars_3
                  )
```

```
                end
                |
                RETURN_0 x =>
                let val (actualsres_4,
                         gensplitexp_4,
                         genseenB4_4,
                         newvars_4
                        ) =
                        sp_do_151 x (^ (no, "4"))
                in  (actualsres_4,
                     RETURN_0 gensplitexp_4,
                     RETURN_0 genseenB4_4,
                     newvars_4
                    )
                end
            )
            p
and sp_do_156 =
    fn p : {2 : exp_0, 1 : string} =>
       fn no =>
          let val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_151 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
                  ) =
                  sp_do_146 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
          in  ((@ (actualsres_2, actualsres_1)),
               (gensplitexp_1, gensplitexp_2),
               (genseenB4_1, genseenB4_2),
               (@ (newvars_2, newvars_1))
              )
          end
and sp_do_155 =
    fn p : {3 : int, 2 : int, 1 : exp_0} =>
       fn no =>
          let val (actualsres_3, gensplitexp_3, genseenB4_3, newvars_3
                  ) =
                  sp_do_143 ((fn {3 = x, ...} => x) p) (^ (no, "3"));
              val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_143 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
                  ) =
                  sp_do_151 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
          in  ((@ (actualsres_3, @ (actualsres_2, actualsres_1))
               ),
               (gensplitexp_1, gensplitexp_2, gensplitexp_3),
               (genseenB4_1, genseenB4_2, genseenB4_3),
               (@ (newvars_3, @ (newvars_2, newvars_1)))
              )
          end
and sp_do_151 =
    fn p : exp_0 =>
       fn no =>
          (fn NUM_0 x =>
              let val (actualsres_1,
                       gensplitexp_1,
                       genseenB4_1,
                       newvars_1
```

```
                    ) =
                    sp_do_143 x (^ (no, "1"))
                in  (actualsres_1,
                     NUM_0 gensplitexp_1,
                     NUM_0 genseenB4_1,
                     newvars_1
                    )
                end
                |
                OP_0 x =>
                let val (actualsres_2,
                         gensplitexp_2,
                         genseenB4_2,
                         newvars_2
                        ) =
                        sp_do_152 x (^ (no, "2"))
                in  (actualsres_2,
                     OP_0 gensplitexp_2,
                     OP_0 genseenB4_2,
                     newvars_2
                    )
                end
                |
                VAR_0 x =>
                let val (actualsres_3,
                         gensplitexp_3,
                         genseenB4_3,
                         newvars_3
                        ) =
                        sp_do_146 x (^ (no, "3"))
                in  (actualsres_3,
                     VAR_0 gensplitexp_3,
                     VAR_0 genseenB4_3,
                     newvars_3
                    )
                end
            )
            p
and sp_do_152 =
    fn p : {2 : explist_0, 1 : string} =>
       fn no =>
          let val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_153 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
                  ) =
                  sp_do_146 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
          in  ((@ (actualsres_2, actualsres_1)),
               (gensplitexp_1, gensplitexp_2),
               (genseenB4_1, genseenB4_2),
               (@ (newvars_2, newvars_1))
              )
          end
and sp_do_153 =
    fn p : explist_0 =>
       fn no =>
          (fn EXPCONS_0 x =>
              let val (actualsres_1,
                       gensplitexp_1,
```

```
                   genseenB4_1,
                   newvars_1
                  ) =
                  sp_do_154 x (^ (no, "1"))
              in  (actualsres_1,
                   EXPCONS_0 gensplitexp_1,
                   EXPCONS_0 genseenB4_1,
                   newvars_1
                  )
              end
              |
              EXPNIL_0 => (nil, EXPNIL_0, EXPNIL_0, nil)
          )
          p
and sp_do_154 =
    fn p : {2 : explist_0, 1 : exp_0} =>
       fn no =>
          let val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_153 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
                  ) =
                  sp_do_151 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
          in  ((@ (actualsres_2, actualsres_1)),
               (gensplitexp_1, gensplitexp_2),
               (genseenB4_1, genseenB4_2),
               (@ (newvars_2, newvars_1))
              )
          end
and sp_do_144 =
    fn p : store_0 =>
       fn no =>
          (fn STORECONS_0 x =>
              let val (actualsres_1,
                   gensplitexp_1,
                   genseenB4_1,
                   newvars_1
                  ) =
                  sp_do_145 x (^ (no, "1"))
              in  (actualsres_1,
                   STORECONS_0 gensplitexp_1,
                   STORECONS_0 genseenB4_1,
                   newvars_1
                  )
              end
              |
              STORENIL_0 => (nil, STORENIL_0, STORENIL_0, nil)
          )
          p
and sp_do_145 =
    fn p : {3 : store_0, 2 : oneexp, 1 : string} =>
       fn no =>
          let val (actualsres_3, gensplitexp_3, genseenB4_3, newvars_3
                  ) =
                  sp_do_144 ((fn {3 = x, ...} => x) p) (^ (no, "3"));
              val (actualsres_2, gensplitexp_2, genseenB4_2, newvars_2
                  ) =
                  sp_do_147 ((fn {2 = x, ...} => x) p) (^ (no, "2"));
              val (actualsres_1, gensplitexp_1, genseenB4_1, newvars_1
```

```
                    ) =
                    sp_do_146 ((fn {1 = x, ...} => x) p) (^ (no, "1"))
            in  ((@ (actualsres_3, @ (actualsres_2, actualsres_1))
                 ),
                 (gensplitexp_1, gensplitexp_2, gensplitexp_3),
                 (genseenB4_1, genseenB4_2, genseenB4_3),
                 (@ (newvars_3, @ (newvars_2, newvars_1)))
                )
            end
    and sp_do_147 =
        fn p : oneexp =>
           fn no =>
              let val newvar = fresh_var' "sp_do_147_" no
              in  (:: (p, nil),
                   ATEXP1exp (VAR1atexp (newvar)),
                   blankcode,
                   :: (newvar, nil)
                  )
              end
    and sp_do_146 = fn p : string => fn no => (nil, p, p, nil)
    and sp_do_143 = fn p : int => fn no => (nil, p, p, nil)
    and sp_do_142 =
        fn p : oneexp =>
           fn no =>
              let val newvar = fresh_var' "sp_do_142_" no
              in  (:: (p, nil),
                   ATEXP1exp (VAR1atexp (newvar)),
                   blankcode,
                   :: (newvar, nil)
                  )
              end;
;
val rec nth_command_0 =
        fn unique_216_0 =>
           (fn (1, PGMCONS_0 (c_0, pgm_1)) => c_0
               |
               (n_0, PGMCONS_0 (c_1, pgm_2)) =>
               nth_command_0 (- (n_0, 1), pgm_2)
               |
               (n_1, PGMNIL_0) => GOTO_0 (~1)
           )
           unique_216_0;
val rec lookup_0 =
        fn unique_217_0 =>
           (fn (x_0, STORENIL_0) => lift_int ~1
               |
               (x_1, STORECONS_0 (x'_0, v'_0, store_1)) =>
               (fn true => ATEXP1exp (PAR1atexp (v'_0))
                   |
                   false => lookup_0 (x_1, store_1)
               )
               (= (x_1, x'_0))
           )
           unique_217_0;
val rec update_0 =
        fn unique_218_0 =>
           (fn (STORENIL_0, x_2, v_0) =>
               STORECONS_0 (x_2, ATEXP1exp (PAR1atexp (v_0)), STORENIL_0)
               |
```

```
              (STORECONS_0 (x'_1, v'_1, s_0), x_3, v_1) =>
              (fn true =>
                  STORECONS_0 (x_3, ATEXP1exp (PAR1atexp (v_1)), s_0)
                  |
                  false =>
                  let val s'_0 =
                          update_0
                          (s_0, x_3, ATEXP1exp (PAR1atexp (v_1)))
                  in  STORECONS_0
                      (x'_1, ATEXP1exp (PAR1atexp (v'_1)), s'_0)
                  end
              )
              (= (x_3, x'_1))
          )
          unique_218_0;

val code = new_code {}; val getcode = fn {} => SEQ1dec (getcode {}, ! code);
val sp_1_seenB4list = ref (nil : ({1 : oneexp}*string) list);
val sp_2_seenB4list = ref (nil : ({1 : oneexp}*string) list);
val rec eval_0 =
        fn unique_219_0 =>
          (fn (NUM_0 n_2, s_1) => lift_int n_2
              |
              (VAR_0 x_4, s_2) => lookup_0 (x_4, s_2)
              |
              (OP_0 ("+", EXPCONS_0 (e1_0, EXPCONS_0 (e2_0, EXPNIL_0))), s_3
              ) =>
              (mk_op "+")
              (ATEXP1exp
               (RECORD1atexp
                (Some
                 (EXPROW1
                  (Lab.mk_IdentLab "1",
                   (eval_0 (e1_0, s_3)),
                   Some
                   (EXPROW1
                    (Lab.mk_IdentLab "2", (eval_0 (e2_0, s_3)), None)
                   )
                  )
                 )
                )
               )
              )
              |
              (OP_0 ("-", EXPCONS_0 (e1_1, EXPCONS_0 (e2_1, EXPNIL_0))), s_4
              ) =>
              (mk_op "-")
              (ATEXP1exp
               (RECORD1atexp
                (Some
                 (EXPROW1
                  (Lab.mk_IdentLab "1",
                   (eval_0 (e1_1, s_4)),
                   Some
                   (EXPROW1
                    (Lab.mk_IdentLab "2", (eval_0 (e2_1, s_4)), None)
                   )
                  )
                 )
```

```
              )
             )
            )
            |
            (OP_0 ("=", EXPCONS_0 (e1_2, EXPCONS_0 (e2_2, EXPNIL_0))), s_5
            ) =>
            sp_1
            ((mk_op "=")
             (ATEXP1exp
              (RECORD1atexp
               (Some
                (EXPROW1
                 (Lab.mk_IdentLab "1",
                  (eval_0 (e1_2, s_5)),
                  Some
                  (EXPROW1
                   (Lab.mk_IdentLab "2", (eval_0 (e2_2, s_5)), None)
                  )
                 )
                )
               )
              )
             )
            )
            |
            (OP_0 ("<", EXPCONS_0 (e1_3, EXPCONS_0 (e2_3, EXPNIL_0))), s_6
            ) =>
            sp_2
            ((mk_op "<")
             (ATEXP1exp
              (RECORD1atexp
               (Some
                (EXPROW1
                 (Lab.mk_IdentLab "1",
                  (eval_0 (e1_3, s_6)),
                  Some
                  (EXPROW1
                   (Lab.mk_IdentLab "2", (eval_0 (e2_3, s_6)), None)
                  )
                 )
                )
               )
              )
             )
            )
            |
            _ => lift_int ~1
        )
        unique_219_0
and sp_2 =
    fn p : oneexp =>
       let val (actualsres, gensplitexp, genseenB4, formalsres) =
               sp_do_142 p "0";
           val abst_actualsres = (nil);
           val abst_formalsres = (nil);
           val formalsresexp_p = vars2exp formalsres;
           val formalsrespat_p = vars2pat formalsres;
           val genseenB4s = {1 = genseenB4};
           val (seen, name) = seenB4 sp_2_seenB4list genseenB4s;
```

```
            val mk_sp_res_pat =
                include_mk_sp_res_pat abst_formalsres formalsrespat_p;
            val sp_res_exp =
                include_sp_res_exp actualsres abst_actualsres;
            val _ =
                (fn true => {}
                    |
                    false =>
                    emit name code
                    (mk_sp_res_fn
                     (::
                      (Some
                       (MRULE1
                        (mk_sp_res_pat
                         (:: ((ATPAT1pat (CON1atpat ("true")), "0"), nil)),
                         (fn _ => lift_int 0) gensplitexp
                        )
                       )
                       handle
                       Match => None,
                       ::
                       (Some
                        (MRULE1
                         (mk_sp_res_pat
                          (:: ((ATPAT1pat (CON1atpat ("false")),"0"), nil)),
                          (fn _ => lift_int 1) gensplitexp
                         )
                        )
                        handle
                        Match => None,
                        nil
                       )
                      )
                     )
                    )
                )
                seen
        in  APP1exp (ATEXP1exp (VAR1atexp (name)), sp_res_exp)
        end
and sp_1 =
    fn p : oneexp =>
        let val (actualsres, gensplitexp, genseenB4, formalsres) =
                sp_do_142 p "0";
            val abst_actualsres = (nil);
            val abst_formalsres = (nil);
            val formalsresexp_p = vars2exp formalsres;
            val formalsrespat_p = vars2pat formalsres;
            val genseenB4s = {1 = genseenB4};
            val (seen, name) = seenB4 sp_1_seenB4list genseenB4s;
            val mk_sp_res_pat =
                include_mk_sp_res_pat abst_formalsres formalsrespat_p;
            val sp_res_exp =
                include_sp_res_exp actualsres abst_actualsres;
            val _ =
                (fn true => {}
                    |
                    false =>
                    emit name code
                    (mk_sp_res_fn
```

```
                        (::
                         (Some
                          (MRULE1
                           (mk_sp_res_pat
                            (:: ((ATPAT1pat (CON1atpat ("true")), "0"), nil)),
                            (fn _ => lift_int 0) gensplitexp
                           )
                          )
                          handle
                          Match => None,
                          ::
                          (Some
                           (MRULE1
                            (mk_sp_res_pat
                             (:: ((ATPAT1pat (CON1atpat ("false")),"0"), nil)),
                             (fn _ => lift_int 1) gensplitexp
                            )
                           )
                           handle
                           Match => None,
                           nil
                          )
                         )
                        )
                       )
                   )
                   seen
           in  APP1exp (ATEXP1exp (VAR1atexp (name)), sp_res_exp)
           end;

val code = new_code {}; val getcode = fn {} => SEQ1dec (getcode {}, ! code);
val sp_3_seenB4list =
    ref (nil : ((int*store_0*pgm_0*int*oneexp)*string) list);
val rec run_0 =
        fn unique_220_0 =>
           (fn (l_0, GOTO_0 n_3, s_7, p_0) =>
               run_0 (n_3, nth_command_0 (n_3, p_0), s_7, p_0)
               |
               (l_1, ASSIGN_0 (x_5, e_0), s_8, p_1) =>
               let val s1_0 = update_0 (s_8, x_5, eval_0 (e_0, s_8))
               in  run_0
                   (+ (l_1, 1), nth_command_0 (+ (l_1, 1), p_1), s1_0, p_1
                   )
               end
               |
               (l_2, COND_0 (e_1, m_0, n_4), s_9, p_2) =>
               sp_3 n_4 s_9 p_2 m_0
               ((mk_op "=")
                (ATEXP1exp
                 (RECORD1atexp
                  (Some
                   (EXPROW1
                    (Lab.mk_IdentLab "1",
                     eval_0 (e_1, s_9),
                     Some (EXPROW1 (Lab.mk_IdentLab "2", lift_int 0, None))
                    )
                   )
                  )
                 )
```

```
            )
           )
           |
           (l_3, RETURN_0 e_2, s_10, p_3) => eval_0 (e_2, s_10)
       )
       unique_220_0
   and sp_3 =
       fn n_4 =>
          fn s_9 =>
             fn p_2 =>
                fn m_0 =>
                   fn p : oneexp =>
                      let val (actualsres_n_4,
                               gensplitexp_n_4,
                               genseenB4_n_4,
                               newvars_n_4
                              ) =
                              sp_do_143 n_4 "1";
                          val (actualsres_s_9,
                               gensplitexp_s_9,
                               genseenB4_s_9,
                               newvars_s_9
                              ) =
                              sp_do_144 s_9 "2";
                          val (actualsres_p_2,
                               gensplitexp_p_2,
                               genseenB4_p_2,
                               newvars_p_2
                              ) =
                              sp_do_148 p_2 "3";
                          val (actualsres_m_0,
                               gensplitexp_m_0,
                               genseenB4_m_0,
                               newvars_m_0
                              ) =
                              sp_do_143 m_0 "4";
                          val n_4 = gensplitexp_n_4
                              and
                              s_9 = gensplitexp_s_9
                              and
                              p_2 = gensplitexp_p_2
                              and
                              m_0 = gensplitexp_m_0;
                          val (actualsres,
                               gensplitexp,
                               genseenB4,
                               formalsres
                              ) =
                              sp_do_142 p "0";
                          val abst_actualsres =
                              (@
                               (actualsres_n_4,
                                @
                                (actualsres_s_9,
                                 @
                                 (actualsres_p_2, @ (actualsres_m_0, nil)
                                 )
                                )
                               )
```

```
    );
val abst_formalsres =
    (@
     (newvars_n_4,
      @
      (newvars_s_9,
       @ (newvars_p_2, @ (newvars_m_0, nil))
      )
     )
    );
val formalsresexp_p = vars2exp formalsres;
val formalsrespat_p = vars2pat formalsres;
val genseenB4s =
    (genseenB4_n_4,
     genseenB4_s_9,
     genseenB4_p_2,
     genseenB4_m_0,
     genseenB4
    );
val (seen, name) =
    seenB4 sp_3_seenB4list genseenB4s;
val mk_sp_res_pat =
    include_mk_sp_res_pat abst_formalsres
    formalsrespat_p;
val sp_res_exp =
    include_sp_res_exp actualsres
    abst_actualsres;
val _ =
    (fn true => {}
        |
        false =>
        emit name code
        (mk_sp_res_fn
         (::
          (Some
           (MRULE1
            (mk_sp_res_pat
             (::
              ((ATPAT1pat (CON1atpat ("true")),"0"),
               nil
              )
             ),
             (fn _ =>
                 run_0
                 (m_0,
                  nth_command_0 (m_0, p_2),
                  s_9,
                  p_2
                 )
             )
             gensplitexp
            )
           )
           handle
           Match => None,
           ::
           (Some
            (MRULE1
             (mk_sp_res_pat
```

```
                                          (::
                                           ((ATPAT1pat (CON1atpat ("false")),
                                             "0"
                                            ),
                                            nil
                                           )
                                          ),
                                          (fn _ =>
                                              run_0
                                              (n_4,
                                               nth_command_0 (n_4, p_2),
                                               s_9,
                                               p_2
                                              )
                                          )
                                          gensplitexp
                                         )
                                        )
                                        handle
                                        Match => None,
                                        nil
                                       )
                                      )
                                     )
                                    )
                                )
                                seen
                       in  APP1exp
                           (ATEXP1exp (VAR1atexp (name)), sp_res_exp
                           )
                       end;
val rec interpret_0 =
        fn unique_221_0 =>
           fn unique_222_0 =>
              (fn ((PGMCONS_0 (c_2, pgm_3)), s_11) =>
                  run_0 (1, c_2, s_11, PGMCONS_0 (c_2, pgm_3))
                  |
                  (PGMNIL_0, _) => lift_int ~1
              )
              (unique_221_0, unique_222_0)

end
```

Appendix C

Ackermann Generating Extension

This is the generating extension for Ackermann's function. The support functions of appendix A are used.

```
(* use "../testprogs/tack.sml.gen"; *)
val _ = use "/home/hugin/birkedal/project/smlmix/sml/testprogs/include.sml";
local
  nonfix + - * / < > <= >= = <> :: ^ @
  open Cogen open OneDecGrammar
in
val rec sp_do_2 =
        fn p : oneexp =>
           fn no =>
              let val newvar = fresh_var' "sp_do_2_" no
              in  (:: (p, nil),
                   ATEXP1exp (VAR1atexp (newvar)),
                   blankcode,
                   :: (newvar, nil)
                  )
              end
    and sp_do_1 = fn p : int => fn no => (nil, p, p, nil);

val code = new_code {}; val getcode = fn {} => SEQ1dec (getcode {}, ! code);
val sp_1_seenB4list = ref (nil : ((int*oneexp)*string) list);
val rec ack_0 =
        fn m_0 =>
           fn n_0 =>
              (fn 0 =>
                  (mk_op "+")
                  (ATEXP1exp
                   (RECORD1atexp
                    (Some
                     (EXPROW1
                      (Lab.mk_IdentLab "1",
                       ATEXP1exp (PAR1atexp (n_0)),
                       Some (EXPROW1 (Lab.mk_IdentLab "2", lift_int 1, None))
                      )
                     )
                    )
                   )
                  )
                  |
                  m'_0 => (sp_1 m'_0) (ATEXP1exp (PAR1atexp (n_0)))
```

```
          )
          m_0
    and sp_1 =
        fn m'_0 =>
           fn p : oneexp =>
              let val (actualsres_m'_0,
                       gensplitexp_m'_0,
                       genseenB4_m'_0,
                       newvars_m'_0
                      ) =
                      sp_do_1 m'_0 "1";
                  val m'_0 = gensplitexp_m'_0;
                  val (actualsres, gensplitexp, genseenB4, formalsres) =
                      sp_do_2 p "0";
                  val abst_actualsres = (@ (actualsres_m'_0, nil));
                  val abst_formalsres = (@ (newvars_m'_0, nil));
                  val formalsresexp_p = vars2exp formalsres;
                  val formalsrespat_p = vars2pat formalsres;
                  val genseenB4s = (genseenB4_m'_0, genseenB4);
                  val (seen, name) = seenB4 sp_1_seenB4list genseenB4s;
                  val mk_sp_res_pat =
                      include_mk_sp_res_pat abst_formalsres
                      formalsrespat_p;
                  val sp_res_exp =
                      include_sp_res_exp actualsres abst_actualsres;
                  val _ =
                      (fn true => {}
                          |
                          false =>
                          emit name code
                          (mk_sp_res_fn
                           (::
                            (Some
                             (MRULE1
                              (mk_sp_res_pat
                               (::
                                ((ATPAT1pat (SCON1atpat (SCon.INTEGER (0))),
                                  "0"
                                 ),
                                 nil
                                )
                               ),
                               (fn _ => ack_0 (- (m'_0, 1)) (lift_int 1))
                               gensplitexp
                              )
                             )
                             handle
                             Match => None,
                             ::
                             (Some
                              (MRULE1
                               (mk_sp_res_pat
                                (::
                                 ((ATPAT1pat (VAR1atpat ("n'_0")), "0"), nil)
                                ),
                                (fn _ =>
                                    ack_0 (- (m'_0, 1))
                                    (ack_0 m'_0
                                     ((mk_op "-")
```

```
                                        (ATEXP1exp
                                         (RECORD1atexp
                                          (Some
                                           (EXPROW1
                                            (Lab.mk_IdentLab "1",
                                             ATEXP1exp (VAR1atexp ("n'_0")),
                                             Some
                                             (EXPROW1
                                              (Lab.mk_IdentLab "2",
                                               lift_int 1,
                                               None
                                              )
                                             )
                                            )
                                           )
                                          )
                                         )
                                        )
                                       )
                                      )
                                  )
                                  gensplitexp
                                 )
                                )
                                handle
                                Match => None,
                                nil
                               )
                              )
                             )
                            )
                        )
                        seen
                in  APP1exp (ATEXP1exp (VAR1atexp (name)), sp_res_exp)
                end
end
```

Bibliography

[And92] Lars Ole Andersen. C program specialization. Technical Report 92/14, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, May 1992.

[And93] Lars Ole Andersen. Binding time analysis and the taming of c pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 47–58. ACM, June 1993.

[BD91] Anders Bondorf and Olivier Danvy. Automatic autoprojection of recursive equations with global variables and abstract data types. *Science of Computer Programming*, 16:151–195, 1991.

[BD93] Anders Bondorf and Dirk Dussart. Handwriting cogen for a CPS-based partial evaluator. Submitted for publication, June 1993.

[BHOS76] Lennart Beckman, Anders Haraldson, Östen Oskarsson, and Erik Sandewall. A partial evaluator and its use as a programming tool. In *Artificial Intelligence 7*, pages 319–357. North-Holland Publishing Company, 1976.

[BJ93] Anders Bondorf and Jesper Jørgensen. Efficient analyses for realistic off-line partial evaluation. To appear in Journal of Functional Programming, 1993.

[BJMS88] Anders Bondorf, Neil D. Jones, Torben Mogensen, and Peter Sestoft. Binding time analysis and the taming of self-application. Draft, 18 pages, DIKU, University of Copenhagen, Denmark, August 1988.

[Blo93] Hendrik Blockell. Een partiële evaluator voor ML. Master's thesis, Katholieke Universiteit Leuven, 1993. In Dutch.

[Bon90] Anders Bondorf. *Self-Applicable Partial Evaluation*. PhD thesis, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, December 1990.

[Bon91] Anders Bondorf. Automatic autoprojection of higher order recursive equations. *Science of Computer Programming*, 17:3–34, March 1991.

[Bon92] Anders Bondorf. Improving binding times without explicit CPS-conversion. In *1992 ACM Conference in Lisp and Functional Programming, San Francisco, California. (Lisp Pointers, vol. V, no. 1, 1992)*, pages 1–10. ACM, 1992.

[Bon93] Anders Bondorf. Similix 5.0 manual. Distributed with the Similix system, 1993.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ML Kit, Version 1. Technical Report 93/14, DIKU, University of Copenhagen, Denmark, 1993. The ML Kit is obtainable by anonymous ftp from `ftp.diku.dk` directory `pub/diku/users/birkedal`. This technical report is distributed along with the ML Kit.

[Con93] Charles Consel. A tour of Schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 145–154. ACM, June 1993.

[dN93] Anne de Niel. *Self-applicable Partial Evaluation of Polymorphically Typed Functional Languages.* PhD thesis, Katholieke Universiteit Leuven, January 1993.

[GJ89] Carsten K. Gomard and Neil D. Jones. Compiler generation by partial evaluation. In G. X. Ritter, editor, *Information Processing '89. Proceedings of the IFIP 11th World Computer Congress*, pages 1139–1144. IFIP, North-Holland, 1989.

[GJ91] Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda-calculus. *Journal of Functional Programming*, 1(1):21–69, January 1991.

[Gom89] Carsten K. Gomard. Higher order partial evaluation — HOPE for the lambda calculus. Master's thesis, DIKU, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, 1989.

[Gom92] Carsten K. Gomard. A self-applicable partial evaluator for the lambda calculus: Correctness and pragmatics. *ACM Transactions on Programming Languages and Systems*, 14(2):147–172, April 1992.

[Hen91] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 448–472. Springer-Verlag, 1991.

[HL92] Carsten Kehler Holst and John Launchbury. Handwriting cogen to avoid problems with static typing. Working Note, October 1992.

[Hol88] Carsten Kehler Holst. Poor man's generalization. Working Note, August 1988.

[Hol89] Carsten Kehler Holst. Syntactic currying. Student report, DIKU, 1989.

[JGS93] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Program Generation.* Prentice-Hall, 1993.

[Jon88] Neil D. Jones. Automatic program specialization: A re-examination from basic principles. In D. Björner, A. P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 225–282. North-Holland, 1988.

[Jon91] Neil D. Jones. Efficient algebraic operations on programs. In *AMAST: Algebraic Methodology and Software Technology*, pages 245–267. University of Iowa, USA, 1991.

[Jon93] Mark P. Jones. Partial evaluation for dictionary-free overloading. Research Report YALEU/DCS/RR-959, Yale University, Department of Computer Science, April 1993.

[Jør92] Jesper Jørgensen. Compiler generation by partial evaluation. Master's thesis, DIKU, Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark, January 1992.

[JSS89] Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[Lau89] John Launchbury. *Projection Factorisations in Partial Evaluation.* PhD thesis, Department of Computing, University of Glasgow, November 1989.

[Lau91] John Launchbury. A strongly-typed self-applicable partial evaluator. In John Hughes, editor, *Functional Programming Languages and Computer Architecture*, Lecture Notes in Computer Science. ACM, Springer-Verlag, 1991.

[Lee89] Peter Lee. *Realistic Compiler Generation.* Foundation of Computing Series. MIT Press, 1989.

[Mey91] U. Meyer. Techniques for partial evaluation of imperative languages. In *Partial Evaluation and Semantics-Based Program Manipulation, New Haven, Connecticut. (Sigplan Notices, vol. 26, no. 9, September 1991)*, pages 94–105. ACM, 1991.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[Mog88] Torben Æ. Mogensen. Partially static structures in a self-applicable partial evaluator. In D. Björner, A. P. Ershov, and Neil D. Jones, editors, *Partial Evaluation and Mixed Computation*, pages 325–347. North-Holland, 1988.

[Mog89a] Torben Æ. Mogensen. *Binding Time Aspects of Partial Evaluation*. PhD thesis, DIKU, University of Copenhagen, Denmark, March 1989. 95 pages.

[Mog89b] Torben Æ. Mogensen. Separating binding times in language specifications. In *Fourth International Conference on Functional Programming Languages and Computer Architecture, London, England, September 1989*, pages 14–25. ACM Press and Addison-Wesley, 1989.

[Mog93] Torben Æ. Mogensen. Constructor specialization. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 22–33, 1993.

[Mos82] Peter D. Mosses. Abstract semantic algebras! In D. Björner, editor, *Formal description of Programming Concepts II*, pages 63–88, Amsterdam, 1982. IFIP IC-2 Working Conference, North Holland.

[Mos93a] Christian Mossin. Partial evaluation of general parsers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 13–21. ACM, June 1993.

[Mos93b] Christian Mossin. Polymorphic binding time analysis. Master's thesis, DIKU, 1993.

[MS92] Morten Marquard and Bjarne Steensgaard. Partial evaluation of an object-oriented imperative language. Master's thesis, DIKU, University of Copenhagen, Denmark, 1992.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NN88] Hanne Riis Nielson and Flemming Nielson. The tml-approach to compiler-compilers. Technical Report 1988–47, Department of Computer Science, Technical University of Denmark, 1988.

[Pag91] Frank G. Pagan. *Partial Computation and the Construction of Language Processors*. Prentice Hall Software Series. Prentice Hall, 1991.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[PS91] Jens Palsberg and Michael I. Schwartzbach. Binding time analysis: Abstract interpretation versus type inference (second revised version). (This article is believed to be unpublished), 1991.

[Rom90] Sergei A. Romanenko. Arity raiser and its use in program specialization. In Neil D. Jones, editor, *ESOP '90. 3rd European Symposium on Programming, Copenhagen, Denmark, May 1990. (Lecture Notes in Computer Science, vol. 432)*, pages 341–360. Springer-Verlag, 1990.

[Ses86] Peter Sestoft. The structure of a self-applicable partial evaluator. In H. Ganzinger and Neil D. Jones, editors, *Programs as Data Objects, Copenhagen, Denmark, 1985. (Lecture Notes in Computer Science, vol. 217)*, pages 236–256. Springer-Verlag, 1986.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, Edinburgh University, Department of Computer Science, Edinburgh University, Mayfield Rd., EH9 3JZ Edinburgh, May 1988. Available as Technical Report CST-52-88.

[Tof90] Mads Tofte. Type inference for polymorphic references. *Information and Compution*, 89(1):1–34, November 1990.

[WCRS91] D. Weise, R. Conybeare, E. Ruf, and S. Seligman. Automatic online partial evaluation. In John Hughes, editor, *Functional Programming Languages and Computer Architecture, Cambridge, Massachusetts, August 1991. (Lecture Notes in Computer Science, vol. 523)*, pages 165–191. ACM, Springer-Verlag, 1991.

[Wel90] Morten Welinder. Disassembler. Student report, DIKU, 1990. Describes the code generation of Turbo Pascal version 3.01A; in Danish.

Index