A Portable Standard ML Implementation

Martin Elsman

The Technical University of Denmark

Department of Computer Science

August 31, 1994

Preface

This master's thesis completes my education at the Technical University of Denmark. The work has been carried out during the period February to August 1994 at the Department of Computer Science under the supervision of Peter Sestoft.

Acknowledgments

I would like to thank my supervisor Peter Sestoft for inspiration, discussions and guidance throughout the period. Also, I would like to thank Lars Birkedal for answering questions regarding the ML Kit system.

Copenhagen
August, 1994

Martin Elsman

Contents

Chapter 1

Introduction

The language Standard ML can be thought of as a strict functional language providing imperative features. The language is naturally split into a core language, that provides many features for programming in the small, and a module language for programming in the large.

Since the first ML compiler was built in 1977 [Pau91, page 1] many other compilers have been implemented. The language Standard ML and its semantics have evolved over a period of about 17 years [MTH90, page 81] with contributions of many people.

Existing implementations of the Standard ML language all have some kind of limitations. The Standard ML of New Jersey system, for example, takes a lot of memory and is cumbersome to port since it produces native machine code. Other implementations have drawbacks such as slow evaluation or in that they fail to evaluate phrases of Standard ML correct as defined in The Definition of Standard ML [MTH90].

At the time of writing new implementations of the Standard ML language are under development. Parallel to my work Sergei Romanenko has developed a core Standard ML compiler that generates byte code and executes it on an abstract machine. This core Standard ML compiler, called Moscow ML, is partly a modification of the Caml Light system with the static elaboration part replaced with the corresponding parts of the ML Kit system (see below). Moscow ML is to a large extend the result of work related to our first attempt to implement a portable Standard ML compiler.

The ML Kit system, that is a Standard ML compiler written in Standard ML and which is very modular, has grown drastically during this period of time. Especially, it is worth mentioning a new back-end for the ML Kit system that uses a *stack of regions* [Tof94]. In this scheme garbage collection can be avoided since allocation and de-allocation of data can be planned statically[1].

There is a need for a portable and small implementation of Standard ML that generates compact code. This report deals with several aspects of the implementation of a compiler for the language Standard ML. The report is split into two parts. The first part describes an attempt to change a front-end of an existing compiler (the Caml Light system) into a Standard ML

[1]This implementation of the new back-end based on *regions* has been developed by Mads Tofte and Lars Birkedal at the University of Copenhagen.

compiler. The result is the $\mathcal{M}ini\mathcal{M}l$ compiler which implements a subset of the core language of Standard ML and which has a module system that supports separate compilation. The $\mathcal{M}ini\mathcal{M}l$ compiler is capable of compiling many small Standard ML example programs, but unfortunately the type checker of the Caml Light system is not *safe*. Also, it does not support built-in overloading and it has no notion of equality types and imperative types. To implement these features would be very time consuming though the algorithms could be adapted from the ML Kit system. It seemed that the entire front-end would have to be substituted (rewritten in Caml Light) with the front-end of the ML Kit system.

The second part of the report is about a new approach. The idea is to construct a new back-end for the ML Kit system. At the time of writing the ML Kit system compiles phrases of core Standard ML into an extended typed lambda language. We show how constructs of this lambda language can be compiled into sequential code that can be executed on an abstract machine. The abstract machine is a modified version of the abstract machine of the Caml Light system. Because of the *high level* instructions (compared to machine instructions) of the Zinc abstract machine one may find that the code generated by such a compiler is very compact.

It is possible for this work to result in a portable version of the ML Kit system by bootstrapping the ML Kit system. Due to inefficiencies in the front end of the ML Kit system this is not possible with the present version of the compiler[2]. Also, the system is not able to translate phrases of the Standard ML module language into constructs of the typed lambda language that is processed by the back-end. At a later stage however, when these inefficiencies and limitations are eliminated, it should be possible to bootstrap the compiler and hence achieve a portable Standard ML compiler that generates compact code.

The first part of the report discusses most parts of the front-end of a core Standard ML compiler. See chapter 2 for a separate introduction. The second part of the report discusses a back-end and a runtime system for a Standard ML compiler. See chapter 8 for a separate introduction.

Whereas the first approach is to replace the *front-end* of an existing compiler (the Caml Light system) the second approach is to replace the *back-end* of an existing compiler (the ML Kit system) to obtain a portable Standard ML implementation that generates compact code.

[2]The version of the ML Kit system that has been used is the 1.0 version with a few extensions (as of April 6, 1994). The lambda language is in this version a typed language and core elaboration is more efficient.

Part I

Mini ML

Chapter 2

The Caml Light System as Point of Departure

In the first part of this report we describe how a compiler for a portable version of a subset of Standard ML, namely $\mathcal{M}ini\mathcal{M}l$ [1], can be developed. The idea is to translate phrases of this subset of Standard ML into binary code that can be evaluated on an abstract machine. The abstract machine is part of the Caml Light system developed at INRIA [Ler93, Ler90b].

The construction of the compiler builds on the bootstrapping capability of the Caml Light system. We change the front-end of the compiler, so that a subset of phrases of Standard ML can be parsed, elaborated and compiled to run on the existing abstract machine of the Caml Light system.

The $\mathcal{M}ini\mathcal{M}l$ compiler is not a Standard ML compiler in that it fails to compile all phrases of the Standard ML core language. Also, the module system that $\mathcal{M}ini\mathcal{M}l$ supports differs from the module language of Standard ML in many ways. The $\mathcal{M}ini\mathcal{M}l$ compiler is written entirely in Caml Light. The compiled byte code is executed on the abstract machine of the Caml Light system, that is written in C.

In the following the different stages of the compilation process will be described.

2.1 Overview of the compiler for $\mathcal{M}_{ini}\mathcal{M}_{l}$

In figure 1 the different steps of compilation are illustrated.

The lexical analysis converts characters to tokens, and it is implemented by use of Caml Lex which is a tool for constructing scanner algorithms, suitable for Caml Light. The lexical analyzer for $\mathcal{M}ini\mathcal{M}l$ is a modification of the lexical analyzer for the Caml Light system. The parser converts correct phrases (sequences of tokens) into an abstract syntax tree. This abstract syntax is basically the abstract syntax of Caml Light, though some additional constructs have been added. The reason is that not all constructs of Standard ML have a corresponding construct in Caml Light, and only a tiny subset of Standard ML can be implemented without

[1] The name $\mathcal{M}ini\mathcal{M}l$ stands for Mini Meta language.

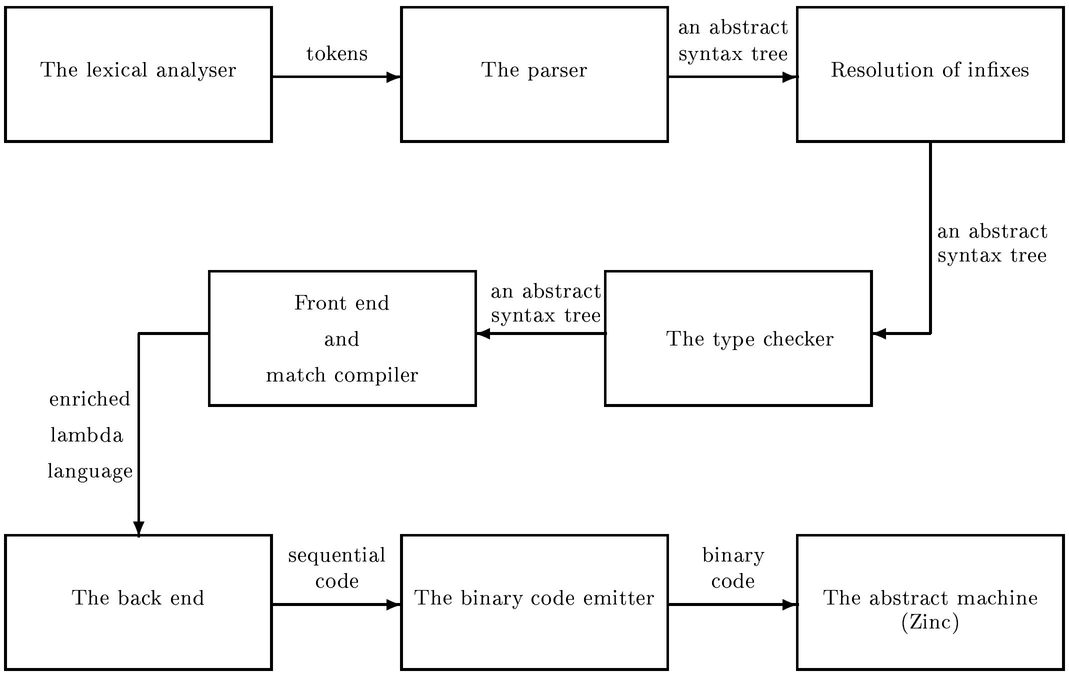


Figure 2.1: Overview of the compiler.

changing the abstract syntax of Caml Light. Infix resolution in Caml Light is handled in a way that does not correspond to the infix resolution of Standard ML, hence it is necessary to introduce a new phase in the compiler for resolving infixes. The method of ML Kit is adopted for this purpose.

The type checker checks that phrases that have been parsed are well typed and it infers types for all expressions. It is also at this stage that overloading of built-in operators in Standard ML is solved. The type checker of Caml Light does not support overloading of built-in operators and it also lacks some other properties that a type checker for Standard ML should have.

It is the job of the match compiler and the front-end to convert the abstract syntax tree into terms in the enriched lambda language. The match compiler eliminates all pattern bindings in the abstract syntax tree producing phrases of the enriched lambda language. Other constructs in the abstract syntax tree are converted into the enriched lambda language by the front-end. The enriched lambda language is translated into sequential code by the *back-end* and then converted into binary code by the *binary code emitter*. The binary code can then be executed on the abstract machine, the Zinc-machine.

Most of the problems arising when converting the Caml Light compiler into a Standard ML compiler are compile time problems. That is, the abstract machine need not be changed radically. Local declarations, such as type, data type and exception declarations are not supported by Caml Light. It is important to notice however, that implementation of such declarations is only a matter of scope and hence a compile time problem[2].

There are however, some runtime problems. Caml Light evaluates expressions right to left[3]

[2]Note that exceptions in Caml Light are not generative as in Standard ML.

[3]In this way curried functions can be implemented very efficiently [Ler90b, page 14]

in contrast to Standard ML, in which expressions are evaluated left to right. This problem can only be solved efficiently by changing the abstract machine. Also, the semantics of equality differs between the Caml Light system and Standard ML and some of the basic operators such as *div* and *mod* behaves differently. These problems relates to the dynamic behavior of $\mathcal{M}ini\mathcal{M}l$.

The module language of Standard ML is very different from the module language of Caml Light. The module language of Caml Light is simpler than that of Standard ML but it has one advantage, namely that it directly supports separate compilation. It is rather easy to adapt the module language of Caml Light for the $\mathcal{M}ini\mathcal{M}l$ system though it is not true to The Definition. The module language of Standard ML however, supports better reusability of code than most other module systems (by parameterization).

In appendix A there is a list of files of source code that have been constructed or altered. Also, there is a description of how to startup the $\mathcal{M}ini\mathcal{M}l$ system.

This part of the report is organized as follows. Lexical analysis and parsing is discussed in chapter 3, and chapter 4 describes how infixes are resolved. In chapter 5 type checking and type inference are studied. In chapter 6 the dynamic aspects of $\mathcal{M}ini\mathcal{M}l$ are discussed. The actual syntax and an informal semantics for the $\mathcal{M}ini\mathcal{M}l$ language are presented in chapter 7.

Chapter 3

Parsing

The lexical analysis of the source code to be compiled converts characters to tokens which are accepted by the parser. The parser converts sequences of tokens into an abstract syntax tree.

3.1 Lexical analysis

As a tool for constructing a lexical scanner Caml Lex [Ler93, pages 113–115], which is a lexical analysis tool for Caml Light, is used. The input file for Caml Lex is basically the scanner for the ML Kit system though it is translated to the style required by Caml Lex. The scanner eliminates comments and recognizes strings within double quotes and several types of constants and identifiers. Reserved names are kept in a hash table; all identifiers found in this table are marked as reserved.

Both keywords that are parts of the core language and keywords that are parts of the module language of Standard ML are delivered to the parser as keywords. To implement the module language of $\mathcal{M}ini\mathcal{M}l$ an additional keyword **close** is needed. A hash table is used to decide efficiently whether a scanned string should be treated as an identifier or a keyword.

3.2 The abstract syntax

Before discussing the parsing phase it is necessary to discuss what is required of the parser. First of all the parsing phase should return an abstract syntax tree for further processing.

In the ML Kit system the abstract syntax, in most cases, directly corresponds to the grammar. When attempting to create an abstract syntax tree that suits the front-end of the Caml Light system, a direct approach cannot be used in every case. The abstract syntax for the ML Kit system is more complicated than the abstract syntax for the Caml Light system, in that it consists of far more levels. As an example, the ML Kit system distinguishes between expressions and atomic expressions in the abstract syntax. This is not the case for the abstract syntax of the Caml Light system. This fact however, will not cause any problems, apart from more complicated code.

Not all parts of the abstract syntax of the Caml Light system are needed. The constructor *Zorpat*, that is a part of the type *pattern_desc* is not needed. Neither are the constructors *Zfor*, *Zvector*, *Zstream* and *Zparser* and the constructors used for dealing with records. These are all parts of the type *expression_desc*. The constructors *Zexpr* and *Zimpldirective*, which are of the type *impl_desc* are not needed either. Parts of the abstract syntax that deals with types and exceptions are moved to another level in the abstract syntax, namely the declaration level. These constructs involve changes in the front end also. Implementing the construct "**let** *dec* **in** *exp* **end**" also requires changes in the abstract syntax. The abstract syntax is split into many datatype constructs, and only the main datatype constructs will be discussed here[1]. Most of the datatype constructs are mutually recursive and depend on types not described here.

A type expression is parsed into one of four constructs.

```
datatype type_expression =
    Typexp of type_expression_desc × location
and type_expression_desc =
    Ztypevar of string
  | Ztypearrow of type_expression × type_expression
  | Ztypetuple of type_expression list
  | Ztypeconstr of global_reference × type_expression list
```

The location information, that is a part of the datatype *type_expression*, is used for error reporting. Error reporting will be discussed in a subsequent section.

Patterns are represented by one datatype construct. There is no distinction between patterns and atomic patterns as in The Definition of Standard ML [MTH90, page 73] or as in the ML Kit system.

```
datatype pattern =
    Pat of pattern_desc × location
and pattern_desc =
    Zwildpat
  | Zvarpat of string
  | Zaliaspat of pattern × string
  | Zconstantpat of atomic_constant
  | Ztuplepat of pattern list
  | Zconstruct0pat of constr_desc global
  | Zconstruct1pat of constr_desc global × pattern
  | Zconstraintpat of pattern × type_expression
  | Zunrespat of pattern list
  | Zunresidentpat of (string list) op_ident_opt
```

Not all patterns are resolved at the stage of parsing. It is necessary to introduce two more constructs representing these unresolved patterns. *Zunrespat* is used to resolve *sequences* of patterns, and *Zunresidentpat* is used to resolve whether an identifier is a constructor or a value.

[1] All parts are shown using Standard ML notation

Expression are represented as shown below. As for patterns no distinction is made between atomic expressions and expressions.

```
datatype expression =
    Expr of expression_desc × location
and expression_desc =
    Zident of expr_ident ref
  | Zconstant of struct_constant
  | Ztuple of expression list
  | Zconstruct0 of constr_desc global
  | Zconstruct1 of constr_desc global × expression
  | Zapply of expression × expression list
  | Zlet of bool × (pattern × expression) list × expression
  | Zfunction of (pattern list × expression) list
  | Ztrywith of expression × (pattern × expression) list
  | Zsequence of expression × expression
  | Zcondition of expression × expression × expression
  | Zwhile of expression × expression
  | Zsequand of expression × expression
  | Zsequor of expression × expression
  | Zconstraint of expression × type_expression
  | Zvector of expression list
  | Zassign of string × expression
  | Zunresexp of expression list
  | Zunresident of (string list) op_ident_opt
  | Zunreslet of declaration list × expression
```

When parsing expressions the infix status of each identifier is not known. Neither is it certain whether an identifier is a constructor or a value. Since it is necessary to delay this resolution two extra constructs, *Zunresexp* and *Zunresident*, are introduced. To be able to parse two declarations without a separating semicolon the construct *Zunreslet* is introduced.

Although the grammar distinguishes between declarations and top level declarations, these constructs are not distinguished in the abstract syntax.

```
and declaration =
    Dec of dec_desc × location
and dec_desc =
    Zvaldef of bool × (pattern × expression) list
  | Ztypedef of (string × string list × type_decl) list
  | Zexcdef of constr_decl list
  | Zinfix of int × string list
  | Zinfixr of int × string list
  | Znonfix of string list
  | Zimpldirective of directiveu
  | Zempty
  | Zunresfun of (pattern list × type_expression
                   list × expression) list list
```

To resolve infixes at a later stage the three kinds of infix declarations

> **infix** ⟨ *d* ⟩ id_1 ⋯ id_n
> **infixr** ⟨ *d* ⟩ id_1 ⋯ id_n
> **nonfix** id_1 ⋯ id_n

are parsed into equivalent constructs. When infixes are resolved the corresponding declaration constructs have been removed from the abstract syntax tree. **fun**-declarations are also solved at the stage of infix resolution, hence it is necessary to introduce the construct *Zunresfun* in the abstract syntax tree. The construct *Zempty* is used to represent empty declarations.

The module language of $\mathcal{M}ini\mathcal{M}l$ allows one to specify a signature (or interface) file, containing a sequence of signature specifications, for each implementation file. A signature specification is parsed into the following construct.

```
datatype intf_phrase =
    Intf of intf_desc × location
and intf_desc =
    Zvaluedecl of (string × type_expression × prim_desc) list
  | Ztypedecl of (string × string list × type_decl) list
  | Zexcdecl of constr_decl list
```

3.3 The grammar

The grammar for Standard ML is described in The Definition of Standard ML [MTH90]. Most of the grammar is given in BNF–notation, but restrictions and some additions are mentioned either in the text or as footnotes. It is necessary to add restrictions to the grammar given in BNF–notation, to eliminate ambiguities.

The grammar for $\mathcal{M}ini\mathcal{M}l$ is given in chapter 7 in BNF–notation and is closely related to the grammar for Standard ML. It builds on the grammar for the ML Kit system, for which most ambiguities are eliminated.

Most of the productions and their actions are straightforward and will not be discussed here. As in Standard ML of New Jersey the implementations of "**val** *ValBind*" and "**val rec** *FnValBind*" are separated to avoid *strange* statements like "**val rec rec rec** ..." [Lab93b, part 3.6].

The expression "**let** *decs* **in** exp_1 ; ... ; exp_i **end**" where *decs* contains more than one declaration, is translated to be equivalent to "**let** dec_1 **in let** dec_2 **in** ... exp_1 ; ... ; exp_i ... **end end**". The semantics for these two kinds of expressions are the same for Standard ML, though this is not mentioned in The Definition of Standard ML [MTH90].

3.4 Elimination of ambiguities

It is not possible to eliminate all the ambiguities in Standard ML by changing the grammar. This problem is solved by accepting a superset of phrases in the language. Then later, in the resolution process or in the actions of the input file for Caml Yacc [Ler93, pages 116–120], the ambiguities are resolved. Phrases that are not acceptable will be detected and then result in an error message. In the following the ambiguities that cannot be eliminated by changing the grammar will be discussed.

Constructors that take arguments must be treated as functions if no arguments are given. That is, it is necessary to detect whether a constructor located in an expression takes an argument or not. Either it takes an argument or it does not take an argument. It is not possible however, at the stage of parsing to determine whether an identifier is a constructor or a variable. For this reason the resolution of identifiers is done at the stage of infix resolution.

In order to parse the pattern construct "$\langle$**op**$\rangle$ *var*$\langle$: *ty*$\rangle$ **as** *pat*" it is necessary to parse the construct "pat_1 **as** pat_2" to avoid introducing an ambiguity in the grammar. When such a pattern is parsed it can easily be checked that pat_1 is indeed a variable. If this is not the case the construct is rejected.

As mentioned earlier several other constructs cannot be resolved at parse-time. Most of these constructs will be discussed in chapter 4.

3.5 Reporting errors

All location information of the source code to be scanned and parsed is handled quite nicely in the Caml Light system. Actions for productions which are similar for the Caml Light system and the ML Kit system appears simpler in the Caml Light system, since the location information is hidden. In Standard ML of New Jersey errors are reported by showing an interpretation of the source code, not by showing the source code itself. It seems to be easier for the user to understand an error when presented the source code, rather than an interpretation of the source code.

Location information is represented by the following construct.

datatype *location* = *Loc* **of** *int* × *int*

The first integer of the constructor tells the position in the corresponding file of the first character of the associated language construct. The second integer tells the position of the character following the associated language construct. During scanning and parsing all language constructs become associated with location information. This can be done easily since location information associated with some sub-constructs of a larger construct is available when this larger construct has been parsed; it is only necessary to deduce the correct location information for the larger construct from location information of the sub-constructs.

Not all syntax errors are caught at the time of parsing. When parsing the construct

"pat_1 **as** pat_2" an exception is raised if pat_1 is not of the form "$\langle$**op**$\rangle var\langle: \ ty\rangle$". This exception is handled in the file compiler.ml together with other error handling exceptions. This however, causes no problems besides from some *syntax errors* appearing later in the source code to be reported earlier. The same is true for errors in sequences of patterns or expressions and function bindings which are checked at the time of infix resolution. In all cases location information is reported to the user.

Chapter 4

Resolving Infixes

It is the job of the parser to create an abstract syntax tree. For many languages it is possible to incorporate the infix status, such as precedence and associativity, of the operators into the grammar for the language. However, when the language becomes as dynamic as Standard ML it is impossible to do this. Standard ML gives the programmer the possibility of redefining an operator, creating new operators and changing the precedence and associativity of an operator that is already defined. For many functional languages which do not give the programmer these possibilities, a resolved abstract syntax tree can be constructed during parsing, using only one pass. In the case of Standard ML however, it is necessary to leave some of the sub trees in the abstract syntax tree unresolved at the time of parsing. This is done simply by creating a node in the abstract syntax tree that includes all the information that is given at parse time. In this way it is possible to resolve this node, the unresolved subtree, when sufficient information about each identifier is available.

The infix resolution technique is adopted from the ML Kit system and builds on the algorithm described in [ASU86, page 203]. Additional constructors are added to the expression and pattern types. During resolution these additional constructors are replaced by constructions that suit the front-end of the compiler, that is the translation of an abstract syntax tree to an extended lambda language construct. As mentioned above the resolution is done by traversing the abstract syntax tree. There are three kinds of nodes in the abstract syntax tree, created by the parser that need to be solved with respect to an environment, containing information about the *fixity* of identifiers. These three kinds of unresolved nodes include a node for unresolved sequences of expressions, a node for unresolved sequences of patterns and a node for unresolved sequences of **fun**–declarations. Nodes that do not contain any of the above unresolved nodes as sub nodes, need not be traversed[1]

An environment containing information about the fixity of identifiers is called an infix basis. To resolve the abstract syntax tree it is necessary to introduce some simple operations on infix bases. These operations include *addition* of an identifier and its infix information to an existing infix basis, and *union* of two infix bases. The infix basis is implemented as a global variable which is updated when a new declaration has been compiled. This compilation might result in additions to the infix basis. In Standard ML the infix basis, in a given scope, can only be

[1]Notice however, that some identifiers still need to be resolved. An identifier is resolved as soon as it can be detected whether it is a constructor or a value.

changed by use of the keywords **infix**, **infixr** and **nonfix**. An infix operator becomes nonfix when prefixed by the keyword **op**, where allowed.

The unresolved abstract syntax constitute a *superset* of what should be included in the resolved abstract syntax. For this reason it is necessary to introduce a new exception to handle errors detected in the infix resolution. This exception is handled in the file compiler.ml as other exceptions used for error handling.

4.1 Expressions and patterns

The method by which the resolution of unresolved sequences of expressions and patterns proceeds is by use of a stack. The input to the resolving functions for expressions and patterns is respectively a list of expressions and a list of patterns. The result of the resolution is respectively an expression and a pattern (resolved nodes in the abstract syntax tree). The stack is used to stack operators and their fixity such that the resulting node with nonfix and infix applications in place can be deduced. To spot applications (two successive operands with no intervening operator) it is necessary to keep track of the last expression respectively pattern parsed in the resolution process.

4.2 Function declarations

The syntax rules of **fun**-bindings are described in The Definition of Standard ML [MTH90, appendix B, fig 20] as a footnote. These rules are formalized below[2]. The parser delivers a **fun**-binding as a sequence of patterns, followed by an optional ": *ty*", and "=" and so on. Of this general syntax we permit the following declarations:

fun *NonfixID NonfixAP+* (: *Ty*) = ...

fun op *ID NonfixAP+* (: *Ty*)? = ...

fun (*NonfixAP InfixID NonfixAP*) *NonfixAP∗* (: *Ty*)? = ...

fun *NonfixAP InfixID NonfixAP* (: *Ty*)? = ...

In the above regular expressions *NonfixID* is any identifier which is not an infix. *InfixID* is an identifier with infix status and *NonfixAP* is any atomic pattern other than an isolated identifier which has infix status. *ID* is any identifier except "=".

[2]This formalization is from The ML Kit code.

Chapter 5

Type Checking

The type checker is the part of the functional programming language implementation that reports to the user information about types of the declared variables and functions. As the name suggests it also checks that the declarations are well–typed. Standard ML is an implicitly and polymorphically typed language. It is an implicitly typed language, since it is optional (in most cases) whether the user should constrain type expressions, and it is polymorphic, since it is possible to define functions that takes arguments of different types.

Basic polymorphic type checking, which is known as *Milner's polymorphic type discipline*, is described in a number of papers, books and articles [Mil78, DM82, Car86, Joh93, Ler92, Jon87, Tof88]. This discipline will be discussed in section 5.1.

Standard ML also provides imperative features such as references to variables. To combine polymorphism with these imperative features is not an easy task [Tof88, Ler92]. It is important though, to provide these features in a functional language since certain algorithms cannot be efficiently implemented otherwise.

5.1 The basic theory of type checking

The ideas illustrated in this section are basically those described in [DM82] and [Car86]. Readers who are familiar with these papers should skip this section. The section is added for completeness and to let the reader become familiar with the notation.

Given a simple applicative language and a syntax of the type system, type inference rules can be defined. It is then possible to infer the type for a given expression in the language [DM82].

5.1.1 The tiny example language

The essence of type checking Standard ML can be explained by type checking a much simpler example language. Also, since most of the language can be built from some basic constructs (e.g. the enriched lambda calculus) we only need to consider a tiny subset of the language. The syntax of the tiny example language follows below.

$$
\begin{array}{rl}
\text{exp} ::= & \text{Id} \\
| & \textit{integer} \\
| & \textit{boolean} \\
| & \text{exp}_1\ \text{exp}_2 \\
| & \textbf{fn}\ \text{Id} \Rightarrow \text{exp} \\
| & \textbf{let val}\ \text{Id} = \text{exp}_1\ \textbf{in}\ \text{exp}_2\ \textbf{end} \\
| & \textbf{(}\ \text{exp},\ \text{exp}\ \textbf{)}
\end{array}
$$

In this language, "Id" is any identifier, and to avoid ambiguities the application $\text{exp}_1\ \text{exp}_2$ associates to the left (as usual).

5.1.2 Typing the tiny example language

Since it is not possible to constrain types to a value or a function explicitly, the example language is purely implicitly typed. It is the task of the type inference to fail if an expression is ill typed and to infer the correct *principal* type, that is the most general type, if an expression is well typed. If a set of *type variables* α and a set of *primitive types* ι (iota), such as integers and booleans, are given, the syntax of *types* τ can be given as

$$
\begin{array}{rl}
\tau ::= & \alpha \\
| & \iota \\
| & \tau \to \tau \\
| & \tau * \tau
\end{array}
$$

It is not sufficient however, to infer a type of an expression simply by unifying a type variable with the types of the expressions that the given expression is associated to. This would be sufficient if no polymorphism were intended. In the construct

```
let
    val identity = fn x => x
in
    (identity true, identity 4)
end
```

it should be possible to apply the function *identity* on arguments of any type. Otherwise polymorphism would be very restricted. This is achieved by using *type schemes* σ:

$$
\begin{array}{rl}
\sigma ::= & \tau \\
| & \forall \alpha \tau
\end{array}
$$

The quantified type variables α in a type scheme $\forall \alpha \tau$ are called *generic* type variables and those that are not quantified are called *non–generic* or *unknowns* [Joh93, page 8] [Jon87, page

172]. A type environment maps every variable name in scope to its type scheme, and whenever a variable goes out of scope it should disappear from the current type environment. Generic type variables can be defined as follows [Car86]:

> *A type variable, occurring in the type of an expression* exp *is generic (with respect to* exp*)* iff *it does not occur in the type of the identifier of any fn-expression enclosing* exp*. That is, when it does not occur free in the type environment.*

When a variable or a function is defined, a type scheme for this variable is introduced or redefined[1] in the environment. Each time a variable is used the type scheme is instantiated such that new type variables are introduced in the type that is associated with the use of the variable.

Not all type variables however, should be instantiated in order for the algorithm not to make *wrong* conclusions. Only generic type variables in a type scheme should be replaced with new type variables during instantiation. Non-generic type variables are simply copied when instantiation takes place. The reason why these two kinds of type variables has to be differentiated can be illustrated by the following example:

$$
\begin{array}{l}
\textbf{let} \\
\quad \textbf{val}\ \mathit{badpair} = \textbf{fn}\ d \Rightarrow (d\ \mathit{true},\ d\ 3) \\
\textbf{in} \\
\quad \ldots \\
\textbf{end}
\end{array}
$$

At first it seems that the abstraction *badpair* can be given the type $(\alpha \rightarrow \beta) \rightarrow (\beta * \beta)$. But then consider applying the badpair to the abstraction $(\textbf{fn}\ n \Rightarrow n + 1)$ which certainly *is* of the type $\alpha \rightarrow \beta$. This will result in applying $(\textbf{fn}\ n \Rightarrow n + 1)$ to *true* hence the type checking algorithm has failed. There are sound extensions of Milner's type system that can type such expressions [Car86] but there seems to be no need of doing so as long as it is possible to type the function *pair* in the example below. In summary , lambda-bound variables do not have their types generalized; only let-bound variables do.

Given a substitution $S = [\tau_i/\alpha_i]$ from type variables α_i to types τ_i, and a type scheme σ, then $S\sigma$ is a new type scheme, an *instance* of σ, where all free occurrences of α_i are replaced by τ_i and where all generic type variables which appear in any τ_i are replaced by new type variables. A generic instance $\sigma' = \forall\beta_1 \ldots \beta_n\tau'$ of a type scheme $\sigma = \forall\alpha_1 \ldots \alpha_m\tau$ is a type scheme where some of the generic type variables in σ have been substituted. We write $\sigma \succ \sigma'$. σ is said to be more general than σ' and it can be shown that iff, for all τ'', whenever $\sigma' \succ \tau''$ then also $\sigma \succ \tau''$ [MTH90, page 19].

5.1.3 Type inference rules for the tiny example language

An assumption $x : \sigma$ maps an identifier to a type scheme. In the following we require that no set of assumptions contains more than one assumption about each identifier. A set of assumptions

[1]If the variable is redefined the type scheme in the environment should be redefined.

is denoted by A. The binary operator $\uplus$ is defined as follows:

$$A_1 \uplus A_2 \equiv \{(id : \sigma) \mid (id : \sigma) \in A_2 \vee ((id : \sigma) \in A_1 \wedge \neg\exists\sigma'.(id : \sigma') \in A_2)\}$$

The operator $\uplus$ overwrites the assumptions in A_1 by those in A_2.

The following type inference rules define what it means for an expression **exp** to be well–typed with type τ under given assumptions A.

$$\begin{array}{rcl}
\mathit{TAUT}: & A \vdash id : \sigma & ((id : \sigma) \text{ in } A) \\[2ex]
\mathit{INST}: & \dfrac{A \vdash exp : \sigma}{A \vdash exp : \sigma'} & (\sigma \succ \sigma') \\[2ex]
\mathit{GEN}: & \dfrac{A \vdash exp : \sigma}{A \vdash exp : \forall\alpha\sigma} & (\alpha \text{ not free in } A) \\[2ex]
\mathit{COMB}: & \dfrac{A \vdash exp : \tau' \to \tau \quad A \vdash exp' : \tau'}{A \vdash (exp\ exp') : \tau} & \\[2ex]
\mathit{ABS}: & \dfrac{A \uplus \{id : \tau'\} \vdash exp : \tau}{A \vdash (\mathbf{fn}\ id \Rightarrow exp) : \tau' \to \tau} & \\[2ex]
\mathit{LET}: & \dfrac{A \vdash exp : \sigma \quad A \uplus \{id : \sigma\} \vdash exp' : \tau}{A \vdash (\mathbf{let\ val}\ id = exp\ \mathbf{in}\ exp'\ \mathbf{end}) : \tau} & \\[2ex]
\mathit{TUP}: & \dfrac{A \vdash exp : \sigma \quad A \vdash exp' : \sigma'}{A \vdash (exp,\ exp') : \sigma * \sigma'} &
\end{array}$$

These inference rules consist of one axiom *TAUT* and a collection of ordinary inference rules. In addition to the inference rules shown in [DM82] the inference rule *TUP* is added to type tuples of two elements. Note that the example language does not provide any mechanism for selecting the first component of a tuple. Built-in functions however, such as $\#1 : \forall\alpha.\forall\beta.\alpha * \beta \to \alpha$ and $\#2 : \forall\alpha.\forall\beta.\alpha * \beta \to \beta$ for selecting a component of a tuple could be provided.

The following example illustrates how the inference rules are used to *prove* that an expression has a given type. To show that the identifier *pair* in the expression

```
let
    val pair =
        let
            val id = fn x => x
        in
            (id true, id 4)
        end
in
    ...
end
```

has type (*bool* $*$ *int*) in the body ($\cdots$) of the **let**-expression, a proof tree is built. It is necessary to split the tree into pieces to make it fit on a page. The proofs of some of the branches follow the main proof. Besides from the rules listed above we implicitly use a *weakening* rule to carry out the proof. The *weakening* rule is defined as:

$$WEAK: \quad \frac{A \vdash exp : \sigma}{B \uplus A \vdash exp : \sigma}$$

The proof is as follows:

$$\text{LET}\ \dfrac{\text{GEN}\ \dfrac{\text{ABS}\ \dfrac{\text{TAUT}\ \dfrac{}{\{x : \alpha\} \vdash x : \alpha}}{\vdash (\mathbf{fn}\ x \Rightarrow x) : \alpha \to \alpha}}{\vdash (\mathbf{fn}\ x \Rightarrow x) : \forall\alpha.\alpha \to \alpha} \qquad \begin{array}{c}\{id : \forall\alpha.\alpha \to \alpha,\ true : bool,\ 4 : int\}\\ \vdash (id\ true, id\ 4) : bool * int\end{array}}{\{true : bool,\ 4 : int\} \vdash (\mathbf{let\ val}\ id = \mathbf{fn}\ x \Rightarrow x\ \mathbf{in}\ (id\ true,\ id\ 4)\ \mathbf{end}) : bool * int}$$

$$\text{TUP}\ \dfrac{\{id : \forall\alpha.\alpha \to \alpha,\ true : bool\} \vdash (id\ true) : bool \qquad \{id : \forall\alpha.\alpha \to \alpha,\ 4 : int\} \vdash (id\ 4) : int}{\{id : \forall\alpha.\alpha \to \alpha,\ true : bool,\ 4 : int\} \vdash (id\ true,\ id\ 4) : bool * int}$$

$$\text{COMB}\ \dfrac{\text{INST}\ \dfrac{\text{TAUT}\ \dfrac{}{\{id : \forall\alpha.\alpha \to \alpha\} \vdash id : \forall\alpha.\alpha \to \alpha}}{\{id : \forall\alpha.\alpha \to \alpha\} \vdash id : bool \to bool} \qquad \text{TAUT}\ \dfrac{}{\{true : bool\} \vdash true : bool}}{\{id : \forall\alpha.\alpha \to \alpha,\ true : bool\} \vdash (id\ true) : bool}$$

$$\text{COMB}\ \dfrac{\text{INST}\ \dfrac{\text{TAUT}\ \dfrac{}{\{id : \forall\alpha.\alpha \to \alpha\} \vdash id : \forall\alpha.\alpha \to \alpha}}{\{id : \forall\alpha.\alpha \to \alpha\} \vdash id : int \to int} \qquad \text{TAUT}\ \dfrac{}{\{4 : int\} \vdash 4 : int}}{\{id : \forall\alpha.\alpha \to \alpha,\ 4 : int\} \vdash (id\ 4) : int}$$

At some points in the proof, though not mentioned, it is necessary to check the additional conditions of the inference rules *INST* and *GEN*. These additional conditions are required to hold whenever *INST* or *GEN* is used in order to make correct conclusions. Note that the polymorphic type (type scheme) $\forall\alpha.\alpha \to \alpha$ is inferred for *id*, and that two different instances are created during type checking, namely $bool \to bool$ and $int \to int$.

It is interesting to notice that in the example above, it is necessary to *guess* the type of the expression $(\mathbf{fn}\ x \Rightarrow x)$ when using the *LET* inference rule. For this reason an algorithm determining the type of an expression that builds directly on such an inference system will be quite inefficient.

5.1.4 An algorithm for type checking

As mentioned it is not easy to apply the inference rules to an arbitrary expression in the language in order to find its type. In the following it will be discussed how an algorithm for the purpose of finding the type of such an arbitrary expression can be constructed.

Instead of guessing a type of x for which a type cannot directly be inferred, the idea is to associate a new type variable, α to x. Whenever x is used and its type is expected to be τ, the equation $\alpha = \tau$ is introduced. To make sure that this equation holds the type variable α and the type τ are *unified*. In general two types, τ and τ' that should have the same type are unified and the unification algorithm produces a substitution S, that maps the free type variables in τ and τ' to types.

To implement an algorithm for finding a type of an arbitrary (valid) expression a unification algorithm is needed. As proposed in [DM82, Rob65] this algorithm U has the following properties:

- Given a pair of types it will either return a substitution V or it will fail.
- If $U(\tau, \tau')$ returns V then V unifies τ and τ' in the sense that $V\tau = V\tau'$. (V is a unifier of τ and τ')
- If S unifies τ and τ' then $U(\tau, \tau')$ returns a substitution V and $\exists R.S = R \circ V$. (V is the most general unifier of τ and τ')
- If $U(\tau, \tau')$ returns V then V will only map type variables involved in τ and τ' (V is the identity on everything else).

Recall that only type variables not free in the assumptions should be generalized (made generic) in "**let val** Id = exp$_1$ **in** exp$_2$ **end**". For this reason it is necessary to define the *closure* of a type τ with respect to assumptions A as follows:

$$\overline{A}(\tau) = \forall \alpha_1 \ldots \alpha_n \tau$$

where $\alpha_1, \ldots, \alpha_n$ are type variables which are free in τ but not in A.

The specification of the algorithm W is written in a *loose* form of Standard ML, meaning that an abstract notation is used when appropriate. It is assumed that the expression that is to be *typed*, is parsed and translated into an abstract syntax tree. This syntax tree is defined, using the notation of Standard ML, as:

```
type idtype = string;

datatype exptype =
    Id of idtype
  | App of exptype × exptype
  | Fn of idtype × exptype
  | Let of idtype × exptype × exptype
  | Tup of exptype × exptype;
```

The algorithm W takes as arguments a set of assumptions A and an abstract representation of an expression (of the type *exptype*). W returns a substitution S and a principal type of the expression specified in the argument. The algorithm W can be defined as:

```
fun W(A, exp) =
    case exp of
        Id (x) ⇒ if $(x{:}\forall\alpha_1 \ldots \alpha_n\tau') \in A$
```

```
            then
                let
                    val $\beta_1 \dots \beta_n = newtypvar()$
                in
                    $([], [\beta_i/\alpha_i]\ \tau')$
                end
            else
                raise $fail$
        | $Int\ (i) \Rightarrow ([], int)$
        | $Bool\ (b) \Rightarrow ([], bool)$
        | $App\ (exp1, exp2) \Rightarrow$
            let
                val $(S_1, \tau_1) = W(A, exp1)$
                val $(S_2, \tau_2) = W(S_1\ A, exp2)$
                val $\beta = newtypvar()$
                val $V = U(S_2\tau_1, \tau_2 \rightarrow \beta)$
            in
                $(V\ S_2S_1, V\beta)$
            end
        | $Fn\ (x, exp1) \Rightarrow$
            let
                val $\beta = newtypvar()$
                val $(S_1, \tau_1) = W(A \uplus \{x : \beta\}, exp1)$
            in
                $(S_1, S_1\beta \rightarrow \tau_1)$
            end
        | $Let\ (x, exp1, exp2) \Rightarrow$
            let
                val $(S_1, \tau_1) = W(A, exp1)$
                val $(S_2, \tau_2) = W(S_1A_x \cup \{x : \overline{S_1A}(\tau_1)\}, exp2)$
            in
                $(S_2S_1, \tau_2)$
            end
        | $Tup\ (exp1, exp2) \Rightarrow$
            let
                val $(S_1, \tau_1) = W(A, exp1)$
                val $(S_2, \tau_2) = W(A, exp1)$
            in
                $(S_2S_1, (S_2\tau_1) \times \tau_2)$
            end;
```

In the algorithm above A_x is defined as:

$$A_x \equiv \{(y : \tau) \in A \mid y \neq x\}$$

The function *newtypvar*() creates a new type variable and the notation S_2S_1 stands for composing the substitutions S_2 and S_1. This substitution has the property:

$$(S_2S_1)\tau = S_2(S_1\tau)$$

$S_2\tau$ stands for applying the substitution S_2 on the type τ and the notation [] above simply stands for the empty substitution, that is $[]\sigma = \sigma$ for all type schemes (identity–operation.)

The typing of the *let*–construct requires an explanation. To implement polymorphism it is necessary to generate a type scheme for the identifier x. In this type scheme *all* type variables $\alpha_1 \ldots \alpha_n$ occurring in τ_1 but not in S_1A should be *generalized.* For this reason the closure $\overline{S_1A}(\tau_1)$ is computed. Type variables which occur in τ_1 but are not generalized, corresponds to non–generic type variables introduced on a higher level, since such type variables will occur both in S_1A and in τ_1. Notice that if non–generic type variables are detected when building the closure $\overline{S_1A}(\tau_1)$ then the construct must be a subexpression of an expression of the form "**fn** Id ⇒ exp".

A polymorphic type inference algorithm, as the one described here, when applied to purely applicative languages can be proved to be sound in the sense that it does not make any *wrong* conclusions [DM82, Mil78]. The type scheme derived by the algorithm is a *principal* type scheme. Every other type scheme of the same expression is a generic instance of the type scheme computed by the algorithm W. It can also be proved to be complete [DM82] in the sense that every derivable type scheme will be an instance of, that is – at least as *specific* as, the type scheme produced by the algorithm W.

5.1.5 An efficient type checking algorithm

Implementations of type checking algorithms based directly on the theory illustrated in the previous section turn out to be bottlenecks in many compilers. The reason is that it seems necessary to handle large environments in a way that is not efficient. In this section however, it will be shown that it is possible to handle these environments in a quite efficient way using *levels*[2].

Consider the tiny example language from section 5.1.1. Besides from a unique name each type variable is also associated with a *let*–level. The idea is that when type checking the expression

$$\textbf{fn}\ x \Rightarrow\ \ exp$$

then x is bound to a fresh type variable for which the level is set to *current* level. When type checking the expression

$$\textbf{let val}\ x = exp_1\ \textbf{in}\ exp_2\ \textbf{end}$$

the **let**–level is increased by one when checking the expression exp_1 and then decreased again. Now, all the type variables in the type τ_1, that is inferred for exp_1, which have an associated level greater than the current level are to be quantified. All type variables in the environment will have lower level; if a type variable has higher level it does not occur in the environment and so it should be generalized. For this to work out correctly it is required that when a type variable α is unified with a type τ then all the levels associated to the type variables for τ together with the type variable α, are to be substituted with the lowest of these levels.

Quantification of a type variable is then done by setting the level of the type variable to -1, e.g. Type instantiation is done by taking a copy of the type where all generic type variables

[2]The idea was presented by Lars Birkedal at The University of Copenhagen, DIKU and described in [R9́2]. The algorithm is implemented in the Caml Light system and in the ML Kit system version 1.x, $x > 0$.

occurring in the type are made fresh (get level equal to present let–level) and where non–generic type variables are "copies" of the type variables of the original type. It is assumed that type variables are represented by references. In this way, when a type variable is unified with a type, the type variable can be updated destructively.

5.2 The type checker for $\mathcal{M}ini\mathcal{M}l$

The type checker for the Caml Light system is not sufficient to type check declarations of Standard ML. A type checker for Standard ML should resolve overloading and also it should allow type variables to be associated with an *equality* attribute and an *imperative* attribute. In $\mathcal{M}ini\mathcal{M}l$ however, the type checker of the Caml Light system is adopted, though it is not sound. The type checker of the Caml Light system is fast, the implementation is rather *small* and it *does* reject most of the phrases that should be rejected by a Standard ML compiler. Because the type checker of $\mathcal{M}ini\mathcal{M}l$ (Caml Light) is not sound it is possible to compile the following sequence of declarations:

```
fun f x =
    let
        val r = ref x
    in
        (fn () => !r, fn k => r := k)
    end;
val (read, write) = f (fn x => x);
val _ = write (fn i => i + 1);
val what = read () true;
```

It is necessary to know of the implementation to predict the result of the last declaration:

$$\textit{val what} = \textit{false} : \textit{bool}$$

The type of the equality operator of $\mathcal{M}ini\mathcal{M}l$ is

$$\forall\, 'a.\, 'a \times\, 'a \to \textit{bool}$$

and not as described in the definition:

$$\forall\, ''a.\, ''a \times\, ''a \to \textit{bool}$$

This is simply because $\mathcal{M}ini\mathcal{M}l$ does not have equality attributes associated with type variables and for this reason it is possible to type check the following declaration:

```
fun f () =
    let
        fun k _ = 0
        fun g _ = 1
    in
        k = g
    end;
```

However, when applying the function f in the above example to a value of type unit, the equality function will raise the exception *Invalid_argument "equal: functional value"*.

Chapter 6

Dynamic Aspects of Mini ML

As mentioned in the introduction it is necessary to change the runtime system of the Caml Light system in order to obtain the behavior that is required of a Standard ML compiler. The aspects that we will discuss here includes order of evaluation and correct implementations of primitives, such as *div*, *mod* and equality ("="). Some parts of this chapter requires knowledge of the abstract machine of the Caml Light system. For information regarding this topic see chapter 10, [Ler90b] and [Ler93, chapter 12].

6.1 Order of evaluation

At the point of writing, $\mathcal{M}ini\mathcal{M}l$ evaluates expressions right to left since it builds on the abstract machine of the Caml Light system. This only shows by use of side effects. The expression

```
let
    val a = ref 0
    fun f x y = !a
in
    f (a := 1) (a := 2)
end;
```

evaluates to

```
val it = 1 : int
```

Not only function applications are evaluated this way; every expression is evaluated from right to left. As an example the expression

```
let
    val a = ref 0
```

```
        val b = (a := 1, a := 2)
    in
        !a
    end;
```

evaluates to

```
    val it = 1 : int
```

and hence shows that also tuples are evaluated from right to left. Similar experiments can be made with lists and other *datatypes.*

When choosing a right–to–left evaluation order, it is possible to evaluate multiple applications very efficiently [Ler90b, page 14]. When evaluating $(M\ N_1\ \ldots\ N_k)$, left to right it seems necessary to reduce M first, then $A_1 = (M\ N_1)$, then N_2, then $A_2 = (A_1\ N_2)$, and so on until $A_k = (A_{k-1}\ N_k)$. Since A_1 has to be computed before N_2 and so on, it is necessary to build the *closures* $A_1, \ldots, A_{k-1}$ during the evaluation process. When evaluating expressions right to left the evaluation order for the above example becomes $N_k, \ldots, N_1, M, A_1, \ldots, A_k$ hence the arguments $N_1, \ldots, N_k$ are available when starting to reduce inside M.

The evaluation order of $\mathcal{M}ini\mathcal{M}l$ can be changed in two ways. One way is to change the lambda-code to Zam-code translation, such that closures are built for every argument (see example above) and such that elements in e.g. tuples are pushed on the stack in the reverse order. If the system is changed in this way the efficient application mechanism of the Zinc–machine will not be used.

The other way of changing the $\mathcal{M}ini\mathcal{M}l$ system such that the evaluation order becomes left to right uses the efficient application mechanism of the Caml Light system. This however, requires a change in the abstract machine. The idea is to introduce an instruction, say *ReverseArgs* (k), in the abstract machine that "reverses" the accumulator and the k top entries on the argument stack[1]:

Code	Accu	Env.	Arg. stack	Return stack
$ReverseArgs(k);c$	a	e	$v_0 \ldots v_{k-1}.s$	r
c	v_{k-1}	e	$v_{k-2} \ldots v_0.a.s$	r

A multiple application is compiled in the Caml Light system as follows:

$$\mathcal{C}\ [\![(M\ N_1 \ldots\ N_k)]\!]\ \equiv Pushmark;\ \mathcal{C}\ [\![N_k]\!]\ ;\ Push;\ \ldots;\ \mathcal{C}\ [\![N_1]\!]\ ;\ Push;\ \mathcal{C}\ [\![M]\!]\ ;\ Apply$$

To obtain left–to–right evaluation the following translation scheme could be used instead:

$$\mathcal{C}\ [\![(M\ N_1 \ldots\ N_k)]\!]\ \equiv Pushmark;\ \mathcal{C}\ [\![M]\!]\ ;\ Push;\ \mathcal{C}\ [\![N_1]\!]\ ;$$

[1]This idea is due to Sergei Romanenko, University of Moscow.

$$\mathit{Push};\ \ldots;\ \mathcal{C}\ [\![N_k]\!]\ ;\ \mathit{ReverseArgs}(k);\ \mathit{Apply}$$

This translation scheme is correct even if functions are not fully applied. To see this, observe that the state of a machine using the right-to-left evaluation scheme and a machine using the left-to-right evaluation scheme (all else equal) is the same prior to the *Apply* instruction. Hence beside from evaluating the function and the arguments in different order the two schemes behave the same.

Still elements in e.g. tuples must be pushed on the stack in reverse order for all expressions to evaluate from left to right.

6.2 Correct implementation of primitives

Not all primitives of the Caml Light system can directly be used in the $\mathcal{M}ini\mathcal{M}l$ system. In order for the dynamic semantics of the primitives in $\mathcal{M}ini\mathcal{M}l$ to match the dynamic semantics of the primitives of The Definition of Standard ML [MTH90, appendix D], it is necessary to change either corresponding primitive operations of the abstract machine, or corresponding primitive functions residing in the Caml Light library. The primitive functions that resides in the Caml Light library can be split into two categories. Some functions are written in the Caml Light language and some are actually written in C using the facilities of Caml Light to link C object code to Caml Light code [Ler93, chapter 12]. Functions that need to be efficient are either C functions or direct operations on the stack and the accumulator in the abstract machine.

Most of the primitives of $\mathcal{M}ini\mathcal{M}l$ behaves semantically correct with respect to The Definition, though some of the primitives do not raise the correct exceptions when required. The abstract machine (the Zinc machine) does not check for overflow on operations on integers though this test could be integrated in the abstract machine. The abstract machine represents an integer i as the value $2 * i + 1$, hence an operation resulting in *overflow* would cause the carry bit to be set[2].

6.2.1 Changing the semantics of equality

The Caml Light system operates with two different notions of equality. One that checks for structural equality and one that checks for physical (referential) equality. In Standard ML there is only one notion of equality. The equality test in Standard ML is basically a structural equality test, though no structural equality is done on references. Equality on two references returns *true* only if the references are identical; otherwise equality on two references returns *false*. In this way no equality test will result in an infinite loop since every loop in a Standard ML data structure either goes through a reference (**ref**) or a *function*. The static semantics of Standard ML requires that no data structure containing functions can be checked for equality. In $\mathcal{M}ini\mathcal{M}l$ however, this is not checked statically but dynamically. That is, an exception

[2]This representation is also used in the Standard ML of New Jersey system [App89, page 5], and this system checks for overflow.

is raised if a data structure containing a function is checked for equality with another data structure.

To incorporate the notion of equality of Standard ML, with respect to references into $\mathcal{M}$*ini*-$\mathcal{M}$*l* it is necessary to make it possible for the runtime system to identify reference cells. This is done by *boxing* all reference values with a special *ref*-tag just as closures, strings and doubles have their own tags. The data in a *ref*-block is then one word denoting a value.

One word			One word
ref–tag	GC	*size* = 1	*value*

One block

The equality function of Caml Light is written in C and it is relatively easy to change this function to match The Definition of Standard ML[3]. When the equality function compares two *ref*-blocks the function returns *true* if the references (pointers) are identical, otherwise *false*. In this way no infinite loops will occur when using the *equal* predicate.

[3]It is of course necessary to recompile the abstract machine and bootstrap the system for the change to appear.

Chapter 7

Using the System

$\mathcal{M}ini\mathcal{M}l$ is an implementation of a subset of the core Standard ML language that is defined in The Definition of Standard ML [MTH90]. The language is built on the basis of another functional language Caml Light, that is developed at INRIA in France[1]. Actually $\mathcal{M}ini\mathcal{M}l$ is a modified version of the Caml Light system, written in Caml Light.

To incorporate many of those features that Standard ML provides, some parts of the ML Kit system, that is a Standard ML written in Standard ML, was translated into Caml Light and integrated with the already existing code. The parser and lexer are translations of code from the ML Kit system.

Due to the module system of Caml Light the system supports separate compilation of modules. The module system of Standard ML does not direct provide such a feature though it gives the user other features such as better reusability of code.

The compiler translates code into binary code that is highly portable. The binary code is executed on an abstract machine, the Zinc machine. The abstract machine itself is written in C and can, for this reason, be transported to many platforms.

The language $\mathcal{M}ini\mathcal{M}l$ is naturally, as Standard ML, split into a core language and a module language.

7.1 The core language

The core language follows the definition of core Standard ML closely. There are however, some constructs of core Standard ML that are not supported in $\mathcal{M}ini\mathcal{M}l$. These constructs will not be mentioned in the grammar (see section 7.1.4).

[1]The Caml Light system is copyright ©1989, 1990, 1991, 1992, 1993 INRIA which holds all ownership rights to the Caml Light system. (See [Ler93, page 5] for more information regarding this topic.)

7.1.1 Reserved words

The reserved words of core $\mathcal{M}ini\mathcal{M}l$ are the same as for core Standard ML [MTH90, page 3] though some of the words has no meaning in $\mathcal{M}ini\mathcal{M}l$. There is one additional reserved word in core $\mathcal{M}ini\mathcal{M}l$ that is not included in The Definition, and that is **close**. Only "=" may be used as an identifier. The reserved words of core $\mathcal{M}ini\mathcal{M}l$ are given below.

abstype	**and**	**andalso**	**as**	**case**	**close**	**do**
datatype	**else**	**end**	**exception**	**fn**	**fun**	**handle**
if	**in**	**infix**	**infixr**	**let**	**local**	**nonfix**
of	**op**	**open**	**orelse**	**raise**	**rec**	**then**
type	**val**	**with**	**withtype**	**while**	(	)
[	]	{	}	,	:	;
...	_	|	=	=>	−>	#

7.1.2 Constants

$\mathcal{M}ini\mathcal{M}l$ supports, as Standard ML, the following tree kinds of special constants (*scon*).

- *integer*: a non–empty sequence of digits, possibly preceded by a negation symbol (`~`). Examples: `23` `~340`.
- *real*: an integer followed by a point (.) and an integer or an integer followed by an exponent or an integer followed by a point (.) and an integer and an exponent. The exponent must consist of an exponent symbol E and an integer. Examples: `4.2` `43.2E32` `~38E~2`.
- *string*: a sequence of printable characters, spaces or escape sequences, enclosed in double–quotes ("). Escape sequences start with a backslash (\) and must be of one of the following forms:

\n	Newline.
\t	Tab.
\^c	Control–c. c may be any character with number 64–95.
ddd	A character with ASCII number *ddd* (the number must be in the interval [0,255]).
\"	"
\\	\
f...*f*\	This sequence is ignored, where *f*...*f* stands for a sequence of spaces, tabs and newlines.

7.1.3 Identifiers

There are six different *classes* of identifiers. These are:

Class	*Description*	*Long*
Var	Value variables	long
Con	Value constructors	long
ExCon	Exception constructors	long
TyVar	Type variables	
TyCon	Type constructors	long
ModId	Module identifier	

As in [MTH90], *var* ranges over Var, *con* over Con and so on. In addition *modid* ranges over ModId. For each class X marked "long" there is a class LongX of *long identifiers*. If x ranges over X then *longx* ranges over LongX. Long identifiers are defined as:

longx ::=	x	identifier
	modid.x	qualified identifier

These long identifiers creates a connection between the core and the modules.

7.1.4 Grammar

The grammar for $\mathcal{M}ini\mathcal{M}l$ is given in BNF–notation. The conventions are as in [MTH90]. The derived forms [MTH90, appendix A] are included in the grammar[2]. The grammar for a program is [MTH90, page 63]:

program ::=	*topdec* ; ⟨*program*⟩	a program

In addition to the phrase classes given in [MTH90, page 7, figure 2] $\mathcal{M}ini\mathcal{M}l$ introduces a new phrase class *TopDec*. This is done to restrict *datatype*, *exception* and *type* declarations from appearing inside *let* declarations.

topdec ::=	**exception** *exbind*	exception declaration
	datatype *datbind*	datatype declaration
	type *typbind*	type declaration
	open *modid*	open declaration
	close *modid*	close declaration
	$topdec_1$; $topdec_2$	sequential toplevel declaration
	dec	declaration

Notice that two *topdec*–declarations need to be separated by a semicolon. In core Standard ML no declarations need to be separated by semicolons.

The grammar for a *standard* declaration follows.

[2]The full grammar for core Standard ML is given in [MTH90, appendix B]

dec ::=	**val** *valbind*	value declaration
	val rec *valbind*	recursive value declaration
	fun *fvalbind*	function declaration
		empty declaration
	$dec_1 \langle ; \rangle dec_2$	sequential declaration
	infix $\langle d \rangle$ $id_1 \cdots id_n$	infix (L) directive, $n \geq 1$
	infixr $\langle d \rangle$ $id_1 \cdots id_n$	infix (R) directive, $n \geq 1$
	nonfix $id_1 \cdots id_n$	nonfix directive, $n \geq 1$
valbind ::=	*pat* = *exp* ⟨**and** *valbind*⟩	
fvalbind ::=	⟨**op**⟩ *var* $atpat_{11} \cdots atpat_{1n}$ $\langle : ty \rangle = exp_1$	$m, n \geq 1$
	$\mid$ ⟨**op**⟩ *var* $atpat_{21} \cdots atpat_{2n}$ $\langle : ty \rangle = exp_2$	See note below
	$\mid$ $\cdots$ $\cdots$	
	$\mid$ ⟨**op**⟩ *var* $atpat_{m1} \cdots atpat_{mn}$ $\langle : ty \rangle = exp_m$ ⟨**and** *fvalbind*⟩	
typbind ::=	*tyvarseq tycon* = *ty* ⟨**and** *typbind*⟩	
datbind ::=	*tyvarseq tycon* = *conbind* ⟨**and** *datbind*⟩	
conbind ::=	⟨**op**⟩ *con* ⟨**of** *ty*⟩ $\langle \mid$ *conbind*⟩	
exbind ::=	⟨**op**⟩ *excon* ⟨**of** *ty*⟩ ⟨**and** *exbind*⟩	
	⟨**op**⟩ *excon* = ⟨**op**⟩ *longexcon* ⟨**and** *exbind*⟩	

In the *fvalbind*–form, if *var* has infix status then either **op** must be present or *var* must be infixed, that is, at the start of any clause the phrase "**op** *var* (*atpat*, *atpat'*) $\cdots$" may be replaced with "(*atpat var atpat'*) $\cdots$", and the parentheses may be dropped if ":*ty*" or "=" follows immediately.

In the **infix**–declaration and the **infixr**–declaration, if the optional *d* is not present, the priority is set to default, that is zero [MTH90, page 6].

The grammar for an expression follows.

exp ::=	*infexp*	
	exp : *ty*	typed (L)
	exp_1 **andalso** exp_2	conjunction
	exp_1 **orelse** exp_2	disjunction
	exp **handle** *match*	handle exception
	raise *exp*	raise exception
	if exp_1 **then** exp_2 **else** exp_3	conditional
	while exp_1 **do** exp_2	iteration
	case *exp* **of** *match*	case analysis
	fn *match*	
infexp ::=	*appexp*	
	$infexp_1$ *id* $infexp_2$	infix expression

appexp ::=	*atexp*	
	appexp atexp	application expression
atexp ::=	*scon*	special constant
	⟨**op**⟩ *longvar*	value variable
	⟨**op**⟩ *longcon*	value constructor
	⟨**op**⟩ *longexcon*	exception constructor
	()	0–tuple
	(exp_1 , ⋯ , exp_n)	n–tuple, $n \geq 2$
	[exp_1 , ⋯ , exp_n]	list, $n \geq 0$
	(exp_1 ; ⋯ ; exp_n)	sequence, $n \geq 2$
	let *dec* **in** exp_1 ; ⋯ ; exp_n **end**	local declaration
	(*exp*)	
match ::=	*mrule* ⟨| *match*⟩	
mrule ::=	*pat* => *exp*	

The *match*–expression extends as far right as possible, hence parentheses may be needed in nested *match*es (e.g. a *case* inside a *case*–branch).

A pattern has the following grammar.

atpat ::=	_	wild-card
	scon	special constant
	⟨**op**⟩ *var*	variable
	⟨**op**⟩ *longcon*	constructor
	⟨**op**⟩ *longexcon*	exception constructor
	()	0–tuple
	(pat_1 , ⋯ , pat_n)	n–tuple, $n \geq 2$
	[pat_1 , ⋯ , pat_n]	list, $n \geq 0$
	(*pat*)	
pat ::=	*atpat*	atomic
	⟨**op**⟩ *longcon atpat*	constructor
	⟨**op**⟩ *longexcon atpat*	exception constructor
	pat_1 *con* pat_2	infixed value construction
	pat_1 *excon* pat_2	infixed exception construction
	pat : *ty*	typed
	⟨**op**⟩ *var* ⟨: *ty*⟩ **as** *pat*	layered

The grammar for type expressions are as follows.

ty ::=	*tyvar*	type variable
	tyseq longtycon	type construction
	ty_1 * ⋯ * ty_n	tuple type, $n \geq 2$
	ty -> *ty′*	function type expression
	(*ty*)	

7.1.5 Phrases of Standard ML not included in $\mathcal{M}ini\mathcal{M}l$

When comparing [MTH90, appendix B] with the above grammar there are a few differences. First of all the above grammar includes the production *topdec*. $\mathcal{M}ini\mathcal{M}l$ does not support *topdec* declarations inside **let** expressions hence it is not possible to nest declarations of types, datatypes or exceptions. Neither does $\mathcal{M}ini\mathcal{M}l$ support two sequential *topdec* declarations not separated with a semicolon. Also records are not supported in $\mathcal{M}ini\mathcal{M}l$. Constructs such as **abstype**– and **withtype**–constructs are not supported either. Also, none of the built–in operators are overloaded.

The lack of these features is a result of not committing to the static semantics of Standard ML. Most of the features could therefore be gained by integrating the type checker of the ML Kit system in the $\mathcal{M}ini\mathcal{M}l$ system.

$\mathcal{M}ini\mathcal{M}l$ evaluates expressions right to left and the exceptions **Neg**, **Quot**, **Prod**, **Sum** and **Diff** are not raised on overflow of the result of arithmetic operations[3]. These topics are discussed in chapter 6 and both have to do with the dynamic semantics of Standard ML.

Streams are not supported in $\mathcal{M}ini\mathcal{M}l$, but could fairly easily be integrated in the system with use of the input/output primitives of the Caml Light system.

7.2 The module system

The module system of $\mathcal{M}ini\mathcal{M}l$ is a C-like module system. The system is able to compile two kind of files – .sml-files (implementation files) and .sig-files (signature files.) The signature files roughly tell what to export from the implementation files. It is not necessary to write signature files for every implementation file. $\mathcal{M}ini\mathcal{M}l$ compiles signature files into .zi-files and implementation files into .zo-files. When $\mathcal{M}ini\mathcal{M}l$ compiles an implementation file it checks if a compiled signature file (.zi-file) exists. If this is not the case $\mathcal{M}ini\mathcal{M}l$ creates one itself.

The grammar for an implementation file simply follows the grammar for the core $\mathcal{M}ini\mathcal{M}l$ language. The name of such files have to end on ".sml". The grammar for a signature file, on the other hand, is a sequence of specifications separated by semicolons. The reserved words for a signature file is a subset of the reserved words for the core $\mathcal{M}ini\mathcal{M}l$ language. The grammar for a signature file follows.

signature ::=	*spec* ; *signature*	specification
		empty specification
spec ::=	**val** *valdesc*	value specification
	type *typdesc*	type specification
	datatype *datdesc*	datatype specification
	exception *excdesc*	exception specification

[3]The Standard ML of New Jersey system raises the exception **Overflow** on overflow of the result of arithmetic operations. That is, the exceptions **Neg**, **Quot**, **Prod**, **Sum** and **Diff** are all equal to the exception **Overflow** [Lab93a, page 13].

valdesc ::=	*id* : *ty* ⟨**and** *valdesc*⟩	value description
	id : *ty* = *d string* ⟨**and** *valdesc*⟩	C function description
typdesc ::=	*tyvarseq tycon* ⟨**and** *typdesc*⟩	type
	tyvarseq tycon = *ty* ⟨**and** *typdesc*⟩	type abbreviation
datdesc ::=	*tyvarseq tycon* = *condesc* ⟨**and** *typdesc*⟩	datatype
condesc ::=	⟨**op**⟩ *id* ⟨**of** *ty*⟩ ⟨| *condesc*⟩	constructor
excdesc ::=	*id* ⟨**of** *ty*⟩ ⟨**and** *excdesc*⟩	exception

The keyword **op** is allowed but has no effect in a *condesc* or an *excdesc*. The Definition requires that **op** should be present when the identifier has infix status [MTH90, page 6]. For a file to be a signature file, the name of the file must end with ".sig".

7.2.1 Intersections with the core language

In *MiniMl* it is possible to access declarations in other files, which must have been compiled, in two ways:

- By use of the **open** and **close** (toplevel) declarations.
- By use of *long* identifiers.

The **open**-declaration takes as argument the name of the file (without extension) to be opened. The **open** declaration does not *overwrite* declarations already declared in a module (file).

7.3 Predefined identifiers and libraries

Predefined identifiers in *MiniMl* constitute a subset of the predefined identifiers of Standard ML. The initial static basis describes the type and the infix status for each identifier [MTH90, appendix C], whereas the initial dynamic basis describes the dynamic semantics for each identifier [MTH90, appendix D]. The lack of overloaded operators makes it necessary to introduce some new names for some of the identifiers involved. In general all identifiers, that have to do with reals and for which there is a counterpart involving integers, are preceded by a %–sign. There is only one exception from this rule. The *MiniMl* function *abs* has type $int \rightarrow int$ and the *MiniMl* function *real_abs*, that has the same *meaning* as the identifier *abs* in The Definition [MTH90, page 75], has type $real \rightarrow real$.

The initial static basis contains the following types:

bool int real string list ref exn unit

The basic value constructors are the identifiers:

true false nil ref ::

The basic exception constructors are:

Chr Div Interupt Mod Ord Match_failure Invalid_argument

Notice that there are no exception constructors for overflow of the result of arithmetic operations.

The following table shows information about each nonfix identifier in the initial static basis.

var	$\mapsto$	σ	*var*	$\mapsto$	σ
map	$\mapsto$	∀'a 'b.('a→'b) → 'a list → 'b list	rev	$\mapsto$	∀'a. 'a list → 'a list
			not	$\mapsto$	bool → bool
~	$\mapsto$	int → int	%~	$\mapsto$	real → real
abs	$\mapsto$	int → int	abs_real	$\mapsto$	real → real
floor	$\mapsto$	real → int	real	$\mapsto$	int → real
sqrt	$\mapsto$	real → real	sin	$\mapsto$	real → real
cos	$\mapsto$	real → real	arctan	$\mapsto$	real → real
exp	$\mapsto$	real → real	ln	$\mapsto$	real → real
size	$\mapsto$	string → int	chr	$\mapsto$	int → string
ord	$\mapsto$	string → int	explode	$\mapsto$	string → string list
implode	$\mapsto$	string list → string	!	$\mapsto$	∀'a. 'a ref → 'a
ref	$\mapsto$	∀'a. 'a → 'a ref	true	$\mapsto$	bool
false	$\mapsto$	bool	nil	$\mapsto$	∀'a. 'a list

Notice the type of *ref*. In The Definition *ref* is given the type ∀ '_a. '_a → '_a ref, where '_a is a *weak* type variable [MTH90, page 75].

The table below contains the type and the infix precedence of each infixed identifier in the initial static basis. All infixed operators in the initial static basis associates to the left except :: and @ that associate to the right[4].

var	$\mapsto$	σ	*var*	$\mapsto$	σ
Precedence 7:					
/	$\mapsto$	real ∗ real → real	div	$\mapsto$	int ∗ int → int
mod	$\mapsto$	int ∗ int → int	∗	$\mapsto$	int ∗ int → int
%∗	$\mapsto$	real ∗ real → real			
Precedence 6:					
+	$\mapsto$	int ∗ int → int	%+	$\mapsto$	real ∗ real → real
−	$\mapsto$	int ∗ int → int	%−	$\mapsto$	real ∗ real → real
^	$\mapsto$	string ∗ string → string			

[4]According to The Definition [MTH90, appendix D] @ should associate to the left. Letting @ associate to the right however, makes multiple appendices more efficient. Apart from the order of evaluation, the result is exactly the same whether it associates to the left or to the right.

Precedence 5:					
::	$\mapsto$	$\forall$ 'a. 'a $*$ 'a list $\to$ 'a list	@	$\mapsto$	$\forall$ 'a. 'a list $*$ 'a list $\to$ 'a list
Precedence 4:					
=	$\mapsto$	$\forall$ 'a. 'a $*$ 'a $\to$ bool	<>	$\mapsto$	$\forall$ 'a. 'a $*$ 'a $\to$ bool
<	$\mapsto$	int $*$ int $\to$ bool	%<	$\mapsto$	real $*$ real $\to$ bool
>	$\mapsto$	int $*$ int $\to$ bool	%>	$\mapsto$	real $*$ real $\to$ bool
<=	$\mapsto$	int $*$ int $\to$ bool	%<=	$\mapsto$	real $*$ real $\to$ bool
>=	$\mapsto$	int $*$ int $\to$ bool	%>=	$\mapsto$	real $*$ real $\to$ bool
Precedence 3:					
:=	$\mapsto$	$\forall$ 'a. 'a ref $*$ 'a $\to$ unit	o	$\mapsto$	$\forall$ 'a 'b 'c. ('b $\to$ 'c) $*$ ('a $\to$ 'b) $\to$ ('a $\to$ 'c)

7.4 The commands

The commands that can be executed from the terminal prompt are the following[5]:

ml	Interactive session.
mlc	Batch compiler and linker.
mlrun	Execution of binary code (.zo–/.zi–files.)
mllibr	The librarian.

The ml command starts an interactive session in which the user can write declarations to be evaluated (see the grammar above.) A *topdec* declaration is evaluated by the system by entering the *topdec* declaration followed by a semicolon and a return [MTH90, page 63].

7.5 Interfacing with C

It is possible to specify a C function in a signature file in $\mathcal{M}ini\mathcal{M}l$. For documentation on how to implement the corresponding C functions, see [Ler93, chapter 12].

[5]The commands are similar to the commands camllight, camlc, camlrun and camllibr of the Caml Light system.

Part II

A New Back-end for the ML Kit System

Chapter 8

The ML Kit System as Point of Departure

The ML Kit system is a Standard ML implementation written in Standard ML [BRTT93]. The ML Kit system version 1.0 provides two different kinds of *back-ends*. There is an interpreter that almost directly corresponds to the sections in The Definition [MTH90, section 4 and 5] describing the dynamic semantics of the core language and the dynamic semantics of the module language. The ML Kit system also includes a *compiler*. The compiler translates abstract syntax trees of the core language into constructs of an extended typed lambda language that can be interpreted by the *lambda language interpreter*. As opposed to the interpreter the compiler (and lambda language interpreter) does not include the module language.

In this part of the report we describe how a new back-end for the ML Kit system is constructed[1]. We show how programs of the typed lambda language of the ML Kit system are compiled into relatively small sequences of byte code that can be executed on an abstract machine.

The ML Kit system is very modular. We describe in chapter 9 how the necessary steps of compilation and execution are integrated with the existing ML Kit system.

The new back-end of the ML Kit system generates code for an abstract machine. This machine is a modified version of the Zinc abstract machine that is a part of the Caml Light system. The abstract machine is described in chapter 10. We describe the representation of values in memory, the changes that have been necessary, how it integrates with the ML Kit system and its limitations.

The translation of the typed lambda language of the ML Kit system into sequential byte code is not done in one pass. Several passes are needed (the compilation by transformation paradigm). The lambda language of the ML Kit system is a typed lambda language based on *unique* names. First we show how this lambda language is translated into a simpler untyped lambda language based on de Bruijn indexes (chapter 11). We then show how this simple lambda language is translated into sequential code (chapter 12) and finally how this sequential code is translated into byte code that can be executed on the abstract machine (chapter 13).

[1]The version of the ML Kit system that has been used is the 1.0 version with a few extensions (as of April 6, 1994). The lambda language is in this version a typed language and core elaboration is more efficient.

Value printing is also naturally done by the abstract machine since the data structures to be printed are only visible to the abstract machine. We show how code for value printing, suitable for the abstract machine, is generated for each value to be printed (chapter 14).

As mentioned above the compiler of the ML Kit system does not support the module language at the time of writing. We discuss what changes are needed in order to compile phrases of the Standard ML module language into the typed lambda language of the ML Kit system, and how these constructs can be translated into sequential byte code (chapter 15).

Chapter 9

Structure of the Implementation

To understand how this new back end to the ML Kit system is structured it is necessary to understand how the ML Kit system itself is structured. The ML Kit system is described in details in [BRTT93], though the version that we work with is somewhat newer. Core *elaboration* has been optimized and the compiler compiles the abstract syntax tree into a *typed* lambda language instead of the untyped lambda language shown in [BRTT93, figure 6.6].

The ML Kit system is highly modular (functorized) which makes it possible to exchange parts of the Standard ML compiler with new parts without too many problems.

9.1 Compilation and evaluation

One part of the ML Kit system that we change is the *evaluation part.* It interacts with the rest of the system through the signature *EVALTOPDEC*.

```
signature EVALTOPDEC =
  sig
    type topdec
    type DynamicBasis
    type Pack

    val RE_RAISE: Pack → unit
    exception UNCAUGHT of Pack
    val pr_Pack: Pack → string

    val eval: DynamicBasis × topdec → DynamicBasis
    val FAIL_USE: unit → unit

    type StringTree
    val layoutDynamicBasis: DynamicBasis → StringTree
  end
```

The function *eval* takes a *dynamic basis* and a *top level declaration* as arguments, evaluates the top level declaration and returns a dynamic basis (additions to the original dynamic basis).

If evaluation causes an uncaught exception the exception *UNCAUGHT*(p) is raised. This exception is then caught by the top level loop. A *dynamic basis* is a collection of environments including a dynamic environment and a tag environment (see below).

The modified functor *CompileAndRun* returns a structure that matches the signature *EVAL-TOPDEC*. It binds together all steps of compilation and the running step. Apart from some basic utility structures it takes as arguments structures that include functions for each step of the compilation together with a structure that includes a function for running byte code. The following steps are applied.

1. Compilation of an abstract syntax tree (of type *topdec*) into a typed lambda program.
2. Optimization of the typed lambda program.
3. Translation of the typed lambda language into a lambda language based on de Bruijn indexes.
4. Compilation of the lambda language based on de Bruijn indexes into sequential code (Zam code).
5. Generation of byte code (Zinc code) from sequential code (Zam code).
6. Running the byte code (Zinc code).

The first and second steps have not been changed and will not be described. A structure that implements the third step should match the signature *TRANSLATE_KIT_LAMBDA*.

```
signature TRANSLATE_KIT_LAMBDA =
  sig
    type DEnv
     and TEnv
     and LambdaPgm
     and dbLambdaPgm
    val lambda_de_bruijn : DEnv × TEnv × LambdaPgm →
                           dbLambdaPgm × TEnv
  end
```

The function *lambda_de_bruijn* takes as arguments a *dynamic environment*, a *tag environment* and a typed lambda program. As a result it returns a lambda program based on de Bruijn indexes and a new tag environment (updates to the original tag environment). The *tag environment* is an environment mapping constructor names to *tags* and *type names* to lists of constructor names. The dynamic environment maps *lambda variables* and *long exception constructors* to *global variables* (indexes to the global store). These environments are both needed for construction of the lambda program based on de Bruijn indexes. See chapter 11 for a detailed description of this step.

A structure that implements the step of compilation of the lambda language based on de Bruijn indexes into sequential code should match the signature *COMPILE_LAMBDA*.

```
signature COMPILE_LAMBDA =
  sig
    type ZamCode
     and dbLambdaPgm
     and DEnv
     and lvar
     and longexcon
     and arity

    val lambda_to_zam : (DEnv × ((lvar × arity) list) ×
                          (longexcon list) × dbLambdaPgm) →
                          ZamCode × DEnv
  end
```

The function *lambda_to_zam* takes as arguments a dynamic environment, lists of lambda variables and long exception constructors (those that should be visible at top level[1], and hence code should be generated to store these *values* in the global store) and a lambda program based on de Bruijn indexes. As a result the function returns a Zam code structure and a new dynamic environment (updates to the original environment). This step is described in details in chapter 12.

A structure that implements the step of generating byte code (Zinc code) from the sequential code (Zam code) should match the signature *EMIT_ZAM*.

```
signature EMIT_ZAM =
    sig
        type ZamCode

        val emit_zam_code : ZamCode → string
        val set_c_primitives : string list → unit
    end
```

The function *emit_zam_code* takes as argument a value of type *ZamCode* and returns a string of byte code. There is also a function *set_c_primitives* used for initialization (see section 10.2). This step is described in details in chapter 13.

A structure that implements the step of running the byte code (Zinc code) should match the signature *RUN_ZINC*.

```
signature RUN_ZINC =
  sig
    val initialize : unit → unit

    exception UNCAUGHT of string
    val run_zinc : string → string

    val terminate : unit → unit
  end
```

[1]For the long exception constructors only the *exception names* (references to strings) should be visible at top level, and hence only these are stored.

The function *initialize* is called when starting an ML Kit session. It starts a separate Unix process in which the abstract machine runs (see section 10.2). The function *run_zinc* takes as argument a string of byte code and returns a string (characters printed on *std_out*). If an *exception* is raised and reaches top level, the exception $UNCAUGHT(p)$ is raised. This exception is *equal* to the exception specified in the signature $EVALTOPDEC$, hence the exception will be caught by the top level loop. There is also a function *terminate* that terminates the abstract machine process.

9.2 Value printing

To print a value stored in the abstract machine, it is necessary to generate Zam code for printing a value of the given type and then execute the code on the Zinc abstract machine.

There is a structure *ValPrint* that provides a function *print* which takes as argument a dynamic basis and the *type* of the value to print. It then generates Zam code to print the value of the given type, translates the Zam code to byte code (Zinc code), executes the byte code on the abstract machine, and receives the result as a string (see above). The structure *ValPrint* is built by the functor *Evaluation* that is a linking functor. The structure that the functor *Evaluation* builds then provides the *print* function in a substructure, and hence it can be used in the top level loop. Value printing is described in details in chapter 14.

Chapter 10

The Abstract Machine

The abstract machine of the Caml Light system – the Zinc machine – is a byte code interpreter which is fairly portable since it is written in C. The abstract machine is described in [Ler90b]. For the Zinc machine to work with the ML Kit system and to obtain the correct dynamic semantics of the Standard ML system, some modifications are needed. Also, not all the machinery of the Zinc machine is necessary.

The Zam (Zinc) abstract machine is basically a call by value Krivine machine [Ler90b, chapter 3], with extensions. It's state has an *accumulator*, an *argument stack*, an *environment*, a *return stack*, a *global store* and of course a pointer to some code. Byte-code that can be directly executed by the Zinc machine is called Zinc code (the actual abstract machine code, see appendix C), in contrast to Zam code which is a sequence of symbolic instructions, where some of the instructions take arguments, such as labels, etc. The semantics of most of the instructions of the Zam abstract machine is given in [Ler90b, chapter 3] and will not be discussed here.

One important topic regarding the abstract machine is how to represent different kinds of data in memory [Ler90b, chapter 4]. This is discussed in section 10.1.

We show in section 10.2 how the abstract machine communicates with the ML Kit system and in section 10.3 we discuss the limits of the Zinc abstract machine.

10.1 Representation of data in memory

A lot of efficiency can be gained by optimizing the representation of data in memory. Representations can be split into two sorts – unboxed representations such as an integer, a value denoting a constructor and so on, and boxed representations that is simply a pointer to a unboxed representation of a value. If we were dealing with a language with no polymorphism, all values could simply be represented by their unboxed representation. This is not possible when dealing with a language that provides polymorphism, since all values have to be of the same *size* (in this case one word). A lot of work has been done regarding optimization of data representations and it turns out that not all data need to be boxed for polymorphism to work [Ler90a, JL92, App89, AM87, App94].

The Zinc abstract machine deallocates dead, unreachable data by use of a garbage collector[1]. When planning the representation of data the garbage collector has to be taken into account. The garbage collector for the Caml Light system is a copying garbage collector that simply runs through (and copies) all valid data to a new address space while adjusting pointers. When all valid data have been copied the old address space can be de-allocated and the new address space used. In such garbage collectors it *must* be possible to distinguish between boxed and unboxed values, for the garbage collector to work.

Some kinds of data need to be distinguishable at runtime (e.g. constructors of a datatype declaration). For this purpose *tagged* values are used. A tagged value is a number of contiguous words in memory.

One word			N words		
tag	GC	*size*	*first word*	...	*N'th word*

One block

The first word is a header that includes a tag (eight bit), some information regarding garbage collection (two bits) and the size (in words) of the data (22 bits). Tagged values with a tag less than the constant *No_scan_tag* (252) are garbage collected. That is, the following words are treated as (pointers to) values, possibly large *data structures*, and not as raw data such as four characters.

We now show how the different kinds of data are represented in memory. Only integers are unboxed. Other values are pointers to allocated objects (tagged values).

10.1.1 Integers

Integers are unboxed. For the garbage collector to distinguish between integers and pointers to other data structures, an integer i is represented as the value $2 * i + 1$, written:

$$\mathcal{A}\,[\![i]\!] \quad\equiv\quad 2 \;*_c\; i \;+_c\; 1$$

Operators with a subscript C are C language operators. Boxed values (pointers to other data structures) are even numbers, hence they have a low-order bit of *zero*. Simple arithmetic operations are not hard with such a representation[2]:

$$\mathcal{A}\,[\![\sim i]\!] \quad\equiv\quad 2 \;-_c\; \mathcal{A}\,[\![i]\!]$$

$$\mathcal{A}\,[\![i \;+\; j]\!] \quad\equiv\quad \mathcal{A}\,[\![i]\!] \;+_c\; \mathcal{A}\,[\![j]\!] \;-_c\; 1$$

[1]This has been thought of as being necessary, but recently it has been discovered that for a strict functional language it is possible to generate code that dynamically allocates and de-allocates data. That is, allocation and de-allocation of data can be planned statically in a strict functional language [Tof94], hence garbage collection can be avoided.

[2]The Standard ML of New Jersey implementation uses a similar representation [App89].

$$\mathcal{A}\llbracket i - j \rrbracket \equiv \mathcal{A}\llbracket i \rrbracket -_c \mathcal{A}\llbracket j \rrbracket +_c 1$$

$$\mathcal{A}\llbracket i * j \rrbracket \equiv 1 +_c ((\mathcal{A}\llbracket i \rrbracket -_c 1) /_c 2) *_c (\mathcal{A}\llbracket j \rrbracket -_c 1)$$

Operations such as **div** and **mod** however, are not as simple. For these operations to work for both positive and negative operands it is necessary to divide the operations into several parts. This is because of the special semantics of these operators [MTH90, page 79]. The following scheme shows how **div** can be defined:

$$\mathcal{A}\llbracket i \ \mathbf{div}\ j \rrbracket \equiv [Div] \qquad j = 0$$

$$\mathcal{A}\llbracket i \ \mathbf{div}\ j \rrbracket \equiv 1 \qquad j \neq 0,\ i = 0$$

$$\mathcal{A}\llbracket i \ \mathbf{div}\ j \rrbracket \equiv 2 *_c ((\mathcal{A}\llbracket i \rrbracket -_c 1) /_c (\mathcal{A}\llbracket j \rrbracket -_c 1)) +_c 1 \qquad i > 0,\ j > 0 \ \vee\ i < 0,\ j < 0$$

$$\mathcal{A}\llbracket i \ \mathbf{div}\ j \rrbracket \equiv 2 *_c ((\mathcal{A}\llbracket i \rrbracket +_c 1) /_c (\mathcal{A}\llbracket j \rrbracket -_c 1)) -_c 1 \qquad i < 0,\ j > 0$$

$$\mathcal{A}\llbracket i \ \mathbf{div}\ j \rrbracket \equiv 2 *_c ((\mathcal{A}\llbracket i \rrbracket -_c 3) /_c (\mathcal{A}\llbracket j \rrbracket -_c 1)) -_c 1 \qquad i > 0,\ j < 0$$

Overflow can be checked for as done when using the simple representation; simply by checking the carry flag dynamically after every integer operation. The range of integers is lower however (only 31 bits available for an integer on a machine with 32 bit words), than if a boxed representation was chosen.

10.1.2 Reals

Reals are *tagged* values with a real-tag that is greater than the constant *No_scan_tag* since the representation is non-structural. The *size* is two words, hence a real value takes up three words in memory:

One word			Two words	
real–tag	GC	*size* = 2	*data*	*data*
			One block	

The words following the header can be directly converted to a C-like double value, hence overflow can be checked easily.

10.1.3 Strings

Strings are *tagged* values. The string-tag need to be greater than the constant *No_scan_tag*, since the representation is non-structural, and the garbage collector should not interpret the string characters as (pointers to) values. The size field is variable and denotes the length of the string in words. The words following the header can be directly converted to a C string, since the string is null-terminated.

10.1.4 Closures

Functional values are represented by closures which are pairs of code pointers and environments. Code is not allocated in the heap but statically allocated and hence not garbage collected. If code were garbage collected as data in the heap it would be necessary for the garbage collector to change return addresses on the return stack, and so the garbage collector would become very complex and slow [Ler90b, page 41].

Closures are, as strings and reals, *tagged* values.

One word			Two words	
closure–tag	GC	*size* = 2	*code pointer*	*environment*

One block

Closures are treated specially by the garbage collector, since the first word proceeding the header should not be collected while the second word (a pointer to a *vector*) should.

10.1.5 Records and tuples

Records and tuples need not be distinguished at run-time, hence they have the same representation. Records are statically sorted with respect to the labels, and the labels are removed from the representation. In this way records and tuples can be represented as zero-*tagged* values where *size* equals the number of fields in the record (or tuple).

10.1.6 Value constructors

There are two kinds of value constructors. Either a constructor takes an argument or it takes no argument (constant constructor). Constant constructors are blocks with *size* zero, hence they only consist of a header. Constructors that take an argument are blocks with *size* one.

Constructors of a given datatype are associated with a unique tag (within the type), such that it is possible to differ between different constructors (of the same type) at run time. Hence the maximal number of different constructors in a given datatype is limited (currently 250).

10.1.7 References

A reference value is basically a constructor *ref* that takes an argument. For the equality test to work correctly at run-time a reference value is given a special tag, such that a reference value can be identified at run time.

10.1.8 Exception names

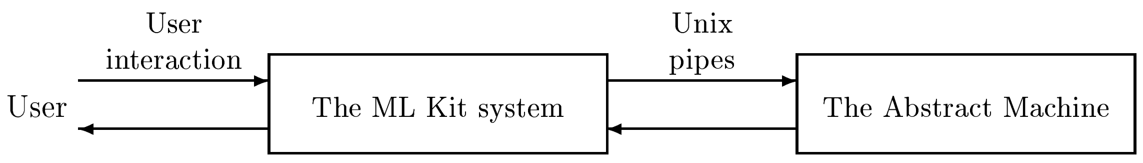
A declaration of an exception causes an exception name to be introduced at runtime. Exception names for constructors taking an argument and exception names for constant constructors have the same representation, namely a reference to a string (the name of the exception constructor, used for printing).

10.1.9 Exception constructors

As for value constructors there are two kinds of exception constructors. All exception constructors are represented in memory as a tuple with two components. For constant constructors the first field of the tuple is an empty place holder, a value of *type* unit. For exception constructors that take an argument the argument (a value) is stored in this field. The second field of the tuple is the exception name (a reference to a string that is simply the name of the exception constructor) [App92, page 49]. Representing exception constructors this way allows us to implement the required generative behavior of exceptions.

10.2 Integrating the ML Kit system and the abstract machine

Currently the ML-Kit system and the abstract machine are two concurrent processes connected by pipes:



The abstract machine is written completely in C. It is a modified version of the abstract machine of the Caml Light system in that some of the operations differ and in that it operates differently. At an earlier stage the initialization code and the execution loop were written in Caml Light, hence this Caml Light program *and* the byte code generated in the ML Kit system were run on the exact same Zinc abstract machine. Since it is necessary to modify some of the basic operations of the Zinc abstract machine to make it correct with respect to The Definition of Standard ML, the initialization process and the execution loop were rewritten in C and integrated with the Zinc abstract machine. Modifying those operations while still using the machine to execute Caml Light programs would be very hard.

When starting the abstract machine, an initialization phase begins. Besides from initializing internal tables etc., the following steps occur:

- Write all available C primitives to *stdout* separated by newlines and terminated by two newlines. Note that the abstract machine's *stdout* is read by the ML Kit process.
- Write the address of the first available global store to *stdout* (four bytes, most significant byte first).

At this point the abstract machine goes into a loop. Each iteration follows the protocol:

- Read the length of the byte code to execute (four bytes) from *stdin*.
  If this length equals zero then terminate.
- Read the number of required global allocations (four bytes) from *stdin*.
- Read the byte code from *stdin*.
- Write result on *stdout* (the result is the printed ASCII representation of the result of the evaluation).
- Write a null character on *stdout*.
- If an exception is raised and not caught by a handler, the name is written to *stdout*.
- Write a null character on *stdout*.

The above protocol has the advantage of being simple, but also has some major disadvantages, as we shall see in chapter 12, since it takes an unacceptable amount of code to introduce string and real constants in the abstract machine. To minimize the size of the code used to introduce string and real constants these constants should be statically allocated and bound to global variables before execution of the byte code. This optimization could easily be adopted in the abstract machine. It would be necessary however, to extend the protocol.

10.3 Limitations of the Zinc abstract machine

There are a few limitations to what can be done when using the Zinc abstract machine of the Caml Light system. Some of these limitations are due to the way data are represented in memory, and other are caused by the way the Zinc abstract machine is built. It is important however, that all these limitations of the representation of data and of the Zinc abstract machine can be eliminated.

The way constructors are represented in memory only allows 250 constructors of a single datatype. This problem however is not severe since only *very* few (in practice none) programs have datatype declarations with more than 250 constructors. If it at some point becomes a problem (perhaps because of code generators) another representation of constructors could be chosen. This choice might cause a lack in performance but the change could easily be worked out.

The number of global variables in the abstract machine is limited to 2^{16}. Only *values* that must be visible on top level are stored in the global store, hence the limitation will cause no problems even for compilation and execution of very large programs (e.g. the ML Kit system itself). If however, it shows that the limit causes problems it would be relatively easy to extend the limit. Only two instructions (*GETGLOBAL* and *SETGLOBAL*) of the Zinc abstract machine need to be changed.

Jumps in the Zinc code are relative *signed* jumps. These jumps are limited to $\pm 2^{15}$ bytes in the code (*small* jumps). This limitation may cause problems when compiling and executing large programs. To extend this limit, in an easy way, all branch instructions of the Zinc abstract machine could be extended to take arguments of four bytes (*long* jumps). It may be possible to allow for both small and long jumps though it requires that the back-patching functions are extended (rewritten).

Blocks in the Zinc abstract machine are limited to 256 fields (0–255). This limit is easily extended by introduction of three new Zinc instructions; one instruction that sets the n'th field of a block for $n > 255$, one that extracts the n'th field of a block for $n > 255$, and one that builds a block of size i, $i > 256$ from the value in the accumulator and the $i - 1$ elements on the argument stack.

Chapter 11

A Lambda Language Based on de Bruijn Indexes

The ML Kit system translates all phrases of core Standard ML into a typed lambda language for which all variables are named. The abstract machine requires sequential code. To translate the typed lambda language into sequential code we first show how the typed lambda language can be translated into a simpler lambda language using de Bruijn indexes as variables.

At this step it is also appropriate to calculate the *size* of each subexpression in the bindings in a *fix*-expression. This can be done using the type information, included in the typed lambda calculus. If we did not calculate these *sizes* at this step we would have to include type information in the target language in order to resolve the *sizes* later. The *size* is the number of machine words needed to represent the value of the expression at runtime.

During this translation phase, constructor *tags* are deduced from the type information given in the typed lambda language. These tags are then introduced in the lambda language based on de Bruijn indexes.

11.1 The source language

The source language, which is given below, is the typed lambda language of the ML Kit system[1].

```
datatype LambdaPgm = PGM of datbinds × LambdaExp
    and datbinds = DATBINDS of
      (tyvar list × TyName × (con × Type Option) list) list list
    and LambdaExp =
          VAR        of {lvar: lvar, instances : Type list}
        | INTEGER    of int
        | STRING     of string
        | REAL       of real
```

[1]The lambda language is the lambda language of ML Kit version 1.0 [BRTT93, page 74], but modified by Lars Birkedal and Mads Tofte to be a typed language.

```
    | FN        of {bound_lvar: {lvar : lvar,
                                 tyvars: tyvar list,
                                 Type: Type},
                   body : LambdaExp}
    | LET       of {bound_lvar: {lvar : lvar,
                                 tyvars: tyvar list,
                                 Type: Type},
                   bind : LambdaExp, scope: LambdaExp}
    | FIX       of {bound_lvars : {lvar : lvar,
                                   tyvars: tyvar list,
                                   Type: Type} list,
                   binds : LambdaExp list, scope : LambdaExp}
    | APP       of LambdaExp × LambdaExp
    | EXCEPTION of excon × Type Option × LambdaExp
    | RAISE     of LambdaExp
    | HANDLE    of LambdaExp × LambdaExp
    | SWITCH_I of int Switch
    | SWITCH_S of string Switch
    | SWITCH_R of real Switch
    | SWITCH_C of longcon Switch
    | SWITCH_E of longexcon Switch
    | PRIM      of Type prim × LambdaExp list
  and 'a Switch = SWITCH of LambdaExp ×
                  ('a × LambdaExp) list × LambdaExp Option
```

The type *datbinds* is a list of groups of mutually recursive datatype bindings while the types *longcon* and *longexcon* are the types for a long constructor and a long exception constructor, respectively. The type *lvar* must be an equality type and the type *primitive* includes constructors for construction and de-construction of records, constructors, and exceptions, together with the *pervasive* functions of Standard ML. Some of these constructs are represented in the following (not complete) datatype:

```
datatype 'Type prim =
      CONprim of {longcon : longcon, instances : 'Type list}
    | DECONprim of {longcon : longcon, instances : 'Type list}
        ⋮
    | PLUS_INTprim
        ⋮
    | EXCONprim of longexcon
    | DEEXCONprim of longexcon
    | RECORDprim
    | SELECTprim of int
```

The type *tyvar* is the type of a type variable and the type *Type* denotes the type of an expression or a lambda variable (lvar):

```
datatype Type =
        TYVARtype   of tyvar
      | ARROWtype   of Type × Type
      | CONStype    of Type list × TyName
      | RECORDtype  of Type list
```

The type *TyName* is the type of the name of a declared type.

11.2 The target language

The target language is very similar to the source language. It is a simple untyped lambda language and instead of *lvars* variables are either de Bruijn indexes (see e.g. [Ses91, page 22]) or indexes to the global store, called *gvars*. The reason for having two kinds of variables is that Standard ML is an interactive language; it should be possible to access values (and exceptions) declared on top-level, earlier in an ML-session.

```
datatype dbLambdaExp =
        dbVAR          of int
      | dbGLOBAL       of gvar
      | dbINTEGER      of int
      | dbSTRING       of string
      | dbREAL         of real
      | dbFN           of dbLambdaExp
      | dbLET          of dbLambdaExp × dbLambdaExp
      | dbFIX          of (dbLambdaExp × int) list × dbLambdaExp
      | dbAPPS         of dbLambdaExp × dbLambdaExp list
      | dbEXCEPTION    of excon × dbLambdaExp
      | dbRAISE        of dbLambdaExp
      | dbHANDLE       of dbLambdaExp × dbLambdaExp
      | dbSWITCH_I     of int dbSwitch
      | dbSWITCH_S     of string dbSwitch
      | dbSWITCH_R     of real dbSwitch
      | dbSWITCH_C     of int dbSwitch
      | dbSTATICFAIL
      | dbPRIM         of db_prim × dbLambdaExp list
  and 'a dbSwitch = dbSWITCH of dbLambdaExp ×
                    ('a × dbLambdaExp) list × dbLambdaExp Option
```

The type *gvar* is basically an address of a variable in the global store. The type *excon* is the type of an exception constructor and the type *db_prim* is a datatype denoting different kinds of primitives. A subset of the primitives (corresponding to those shown for the source language) for the target language follows:

```
datatype db_prim =
        dbCONprim of int
```

| *dbDECONprim*
⋮
| *dbPLUS_INTprim*
⋮
| *dbEXCONprim* **of** *int*
| *dbGLOBALEXCONprim* **of** *gvar*
| *dbDEEXCONprim*
| *dbRECORDprim*
| *dbSELECTprim* **of** *int*

As for variables exception constructors are divided into two kinds. Exception constructors denoted as *dbEXCONprim* are local exception constructors whereas exception constructors denoted as *dbGLOBALEXCONprim* are declared on top-level earlier in an ML-session, and hence global.

11.3 The $\mathcal{T}$ translation scheme

In the translation a compile time environment *ce* is needed to translate lambda variables (LV) and exception constructors (EXC) into de Bruijn indexes. The *ce* environment must be passed to each of the mutually defined functions in the translation process.

There is also need for another environment, the *de* environment. This environment however, can be global to the mutually defined compilation functions, since it is not *altered* during the translation process. The environment *de* is the *dynamic environment* and it includes only values and exceptions, previously declared on top level. This environment maps lambda variables (lvars) and long exception constructors (longexcons) to gvars (really integers) that represents locations (addresses) in the global store. If there were no such dynamic environment it would not be possible to access variables declared on top-level earlier in an ML-session. To look up an address in the dynamic environment, given a lambda variable or a long exception constructor the two lookup-functions *lookup_gvar_from_lvar* and *lookup_gvar_from_longexcon* are given.

Most of the primitives are translated trivially. That is, the representation is the same in both the typed lambda language and the lambda language based on de Bruijn indexes. In the following translation scheme most of the trivially translated primitives are not included. Note however, that it is the job of one of these trivially translated primitives to *store* a variable in the global store.

The translation scheme, $\mathcal{T}$ can now be given. There are two different schemes for variables.

$$\mathcal{T}\ [\![VAR\{lvar = x_i, \ldots\}]\!]\ \overbrace{[d_0, d_1, \ldots, d_{i-1}, LV\ \ x_i, \ldots, d_{n-1}]}^{ce} \equiv$$
$$dbVAR\ i \qquad x_i \notin dom(de)$$

$$\mathcal{T}\ [\![VAR\{lvar = x, \ldots\}]\!]\ ce \equiv$$

$$dbGLOBAL\ (lookup\_gvar\_from\_lvar\ x) \qquad x \in dom(de),\ \ x \notin dom(ce)$$

Constants are translated trivially as shown below.

$$\mathcal{T}\ [\![INTEGER\ n]\!]\ ce \equiv dbINTEGER\ n$$
$$\mathcal{T}\ [\![STRING\ s]\!]\ ce \equiv dbSTRING\ s$$
$$\mathcal{T}\ [\![REAL\ r]\!]\ ce \equiv dbREAL\ r$$

When translating the body of a lambda abstraction the environment *ce* must be extended with the lambda variable bound by the abstraction.

$$\mathcal{T}\ [\![FN\{bound\_lvar = \{lvar = lv, \ldots\}, body = body\}]\!]\ ce \equiv$$
$$dbFN\ (\ \mathcal{T}\ [\![body]\!]\ (LV\ \ lv :: ce)\ )$$

The environment for the scope of a *let*-expression also need to be extended with the lambda variable to be bound in the *let*-binding.

$$\mathcal{T}\ [\![LET\{bound\_lvar = \{lvar = lv, \ldots\}, bind = bind, scope = scope\}]\!]\ ce \equiv$$
$$dbLET\ (\ \mathcal{T}\ [\![bind]\!]\ ce,\ \mathcal{T}\ [\![scope]\!]\ (LV\ \ lv :: ce)\ )$$

When translating a *fix*-expression we first create an environment including all variables to be bound. Each subexpression will then be translated in this environment.

$$\mathcal{T}\ [\![FIX\{bound\_lvars = blvars, binds = binds, scope = scope\}]\!]\ ce \equiv$$

```
let
    val ce’ = (map (fn {lvar=lvar, ...} ⇒ LV lvar) blvars) @ ce
    val types = map (fn {Type=Type, ...} ⇒ Type) blvars
    fun makepairs [] _ = []
      | makepairs (b::bs) (t::ts) = (size_of_type t, $\mathcal{T}$ [[b]] ce’)::
                        makepairs bs ts
      | makepairs _ _ = raise Impossible
    val pairlist = makepairs binds types
    val s = $\mathcal{T}$ [[scope]] ce’
in
    dbFIX (pairlist, s)
end
```

The exception *Impossible* should not be raised since the list of bindings and the list of bound lambda variables have the same length by construction. The function *size_of_type* is defined below and it returns the number of machine words, needed to represent a value of the given type at runtime.

```
fun size_of_type TYVARtype _  = 1
  | size_of_type ARROWtype _  = 2
  | size_of_type CONStype _   = 1
  | size_of_type RECORDtype l = List.size l
```

Translation of an application is trivial.

$$\begin{gathered}\mathcal{T}\ [\![APP(body,\ arg)]\!]\ ce \equiv \\ dbAPP(\mathcal{T}\ [\![body]\!]\ ce,\ \mathcal{T}\ [\![arg]\!]\ ce)\end{gathered}$$

Translations of expressions involving exceptions follow.

$$\begin{gathered}\mathcal{T}\ [\![EXCEPTION(excon,\ None,\ lexp)]\!]\ ce \equiv \\ dbEXCEPTION(excon,\ \mathcal{T}\ [\![lexp]\!]\ (EXC\ \ excon :: ce))\end{gathered}$$

$$\mathcal{T}\ [\![RAISE\ \ lexp]\!]\ ce \equiv dbRAISE(\mathcal{T}\ [\![lexp]\!]\ \text{ce})$$

$$\begin{gathered}\mathcal{T}\ [\![HANDLE(lexp\_body,\ lexp\_handle)]\!]\ ce \equiv \\ dbHANDLE(\mathcal{T}\ [\![lexp\_body]\!]\ ce,\ \mathcal{T}\ [\![lexp\_handle]\!]\ (DUMMY :: ce))\end{gathered}$$

All switch expressions are basically translated the same way. As an example the translation scheme for a constructor switch is as follows.

$$\mathcal{T}\ [\![SWITCH\_C(SWITCH(lexp,\ longcon\_lexp\_list,\ opt))]\!]\ ce \equiv$$

```
let
   val tr_lexp = T [[lexp]] ce
   val tr_list = map (fn (longcon, l) =>
           (get_longcon_tag longcon, T [[l]] ce)) longcon_lexp_list
   val tr_opt =
        case opt of
            Some l => Some (T [[l]] ce)
          | None => None
in
   dbSWITCH_C(dbSWITCH(tr_lexp, tr_list, tr_opt))
end
```

The function *get_longcon_tag* takes as argument a *longcon* and lookups a tag value (really an integer) in the *tag environment* (see chapter 9).

As mentioned earlier most of the translations of primitives are trivial. For some of the primitives however, the translation schemes are stated below.

$$\begin{gathered}\mathcal{T}\ [\![\mathit{PRIM}(\mathit{CONprim},\ \mathit{lexps})]\!]\ \mathit{ce} \equiv \\ \mathit{dbPRIM}\ (\mathit{dbCONprim}(\mathit{get\_longcon\_tag}\ \mathit{longcon}), \\ \mathit{map}\ (\mathbf{fn}\ \mathit{lexp} \Rightarrow \mathcal{T}\ [\![\mathit{lexp}]\!]\ \mathit{ce})\ \mathit{lexps})\end{gathered}$$

$$\begin{gathered}\mathcal{T}\ [\![\mathit{PRIM}(\mathit{DECONprim}\ \_,\ [\mathit{lexp}])]\!]\ \mathit{ce} \equiv \\ \mathit{dbPRIM}\ (\mathit{dbDECONprim},\ [\mathcal{T}\ [\![\mathrm{lexp}]\!]\ \mathit{ce}])\end{gathered}$$

The translation of an exception constructor is twofold. The first translation scheme is used when an exception is declared on top level earlier in the ML-session. The second translation scheme is used when an exception is declared in a local scope. The reason for dividing exception constructors into two cases is the same as for dividing variables into two cases. It should also be possible to *access* exception constructors, declared earlier in an ML-session.

$$\begin{gathered}\mathcal{T}\ [\![\mathit{PRIM}(\mathit{EXCONprim}\ \mathit{longexcon},\ \mathit{lexps})]\!]\ \overbrace{[v_0, v_1, \ldots, v_{i-1}, \mathrm{EXC}\ x_i, \ldots, v_{n-1}]}^{\mathrm{ce}} \equiv \\ \mathit{dbPRIM}(\mathit{dbEXCONprim}\ i,\ \mathit{map}\ (\mathbf{fn}\ \mathit{lexp} \Rightarrow \mathcal{T}\ [\![\mathit{lexp}]\!]\ \mathit{ce})\ \mathit{lexps}) \\ x_i = \mathit{excon\_of\_longexcon}\ \mathit{longexcon},\ \ \mathit{EXC}\ x_i\ \notin \{v_0, v_1, \ldots, v_{i-1}\}\end{gathered}$$

$$\begin{gathered}\mathcal{T}\ [\![\mathit{PRIM}(\mathit{EXCONprim}\ \mathit{longexcon},\ \mathit{lexps})]\!]\ \mathit{ce} \equiv \\ \mathit{dbPRIM}(\mathit{dbGLOBALEXCONprim}(\mathit{lookup\_gvar\_from\_longexcon}\ \mathit{longexcon}), \\ \mathit{map}\ (\mathbf{fn}\ \mathit{lexp} \Rightarrow \mathcal{T}\ [\![\mathit{lexp}]\!]\ \mathit{ce})\ \mathit{lexps}) \\ x = \mathit{excon\_of\_longexcon}\ \mathit{longexcon}, \\ \mathit{longexcon}\ \in \mathit{dom}(\mathit{de}),\quad \mathit{EXC}\ x \notin \mathit{dom}(\mathit{ce})\end{gathered}$$

The function *excon_of_longexcon* extracts the exception constructor from a long exception constructor. If an exception constructor is in the domain of the *ce* environment the first scheme is chosen. This ensures that an exception constructor of a *local* exception declaration, with the same name as the exception constructor for a *global* exception declaration will be *visible.* Also note that since exception constructors (excons) are not unique, as lambda variables in the typed lambda language are, it is required that the *lookup* function of the *ce* environment returns the first instance of any matching exception constructor (the latest one introduced) if any. This ensures that the correct exception constructor is extracted from the environment.

Translations of the primitives for introduction of records and for selecting a sub-term of a record are simple.

$$\begin{gathered}\mathcal{T}\ [\![\mathit{PRIM}(\mathit{RECORDprim},\ \mathit{lexps})]\!]\ \mathit{ce} \equiv \\ \mathit{dbPRIM}\ (\mathit{dbRECORDprim},\ \mathit{map}\ (\mathbf{fn}\ \mathit{lexp} \Rightarrow \mathcal{T}\ [\![\mathit{lexp}]\!]\ \mathit{ce})\ \mathit{lexps})\end{gathered}$$

$$\mathcal{T}\ [\![PRIM(SELECTprim\ n,\ [lexp])]\!]\ ce \equiv$$
$$dbPRIM\ (dbSELECTprim\ n,\ [\mathcal{T}\ [\![lexp]\!]\ ce])$$

In the above translation schemes *map* is defined as usual and the compile time environment *ce* is represented as a list. The *lookup* function is given below.

```
fun lookup ([], _) = raise Unbound_lvar
  | lookup (y :: yr, x) = if x = y then 0
                          else 1 + lookup (yr, x)
```

Representing the environment *ce* this way causes the indexes of all variables of the list to be updated, as required, when adding a variable to the environment. Each entry in the *ce* environment is either a lambda variable (LV), an exception constructor (EXC) or a dummy variable ($DUMMY$):

```
datatype ce_entry =
        LV of lvar
      | EXC of excon
      | DUMMY
```

To improve efficiency this transformation step can be integrated in the pass where the lambda language based on de Bruijn indexes is compiled into Zam instructions, but this has not been done for clarity reasons.

Chapter 12

Generating Sequential Code

In this chapter we show how the simple lambda language based on de Bruijn indexes is compiled into sequential code. The syntax of the lambda language based on de Bruijn indexes was presented in chapter 11. The syntax of the sequential code is a modification of the syntax of the sequential code (Zam code) used in the Caml Light system.

The sequential code is split into three parts. These are *initial* code, *function* code and *binding* code. The binding code binds all variables that should be visible on top level to global variables. To generate the binding code however, it is necessary to know which variables should be visible and where these variables should be stored. Generation of binding code is discussed in section 12.3. The reason for dividing function code and initial code is that "jumping around functions" can be avoided this way. Generation of initial code and function code is discussed in section 12.2.

12.1 Syntax of the sequential code

As mentioned above the sequential code is split into three parts each containing a list of Zam instructions.

$$\textbf{datatype}\ \mathit{ZamCode} = \mathit{ZAMCODE}\ \textbf{of}\ \{\begin{array}{l} \mathit{init\_code} : \mathit{zam\_instruction\ list}, \\ \mathit{bind\_code} : \mathit{zam\_instruction\ list}, \\ \mathit{functions} : \mathit{zam\_instruction\ list}\} \end{array}$$

A Zam instruction follows the syntax below.

$$\begin{array}{l} \textbf{datatype}\ \mathit{zam\_instruction} = \\ \quad\quad \mathit{Kquote}\ \textbf{of}\ \mathit{struct\_constant} \\ \quad\ \mid \mathit{Kget\_global}\ \textbf{of}\ \mathit{gvar} \mid \mathit{Kset\_global}\ \textbf{of}\ \mathit{gvar} \\ \quad\ \mid \mathit{Kaccess}\ \textbf{of}\ \mathit{int} \mid \mathit{Kgrab} \\ \quad\ \mid \mathit{Kpush} \mid \mathit{Kpop} \mid \mathit{Kpushmark} \\ \quad\ \mid \mathit{Klet} \mid \mathit{Kendlet}\ \textbf{of}\ \mathit{int} \end{array}$$

```
          | Kapply | Ktermapply
          | Kcheck_signals
          | Kreturn | Kclosure of int
          | Kletrec1 of int
          | Kmakeblock of constr_tag × int
          | Kprim of primitive
          | Kpushtrap of int | Kpoptrap
          | Klabel of int
          | Kbranch of int | Kbranchif of int | Kbranchifnot of int
          | Kstrictbranchif of int | Kstrictbranchifnot of int
          | Ktest of bool_test × int
          | Kbranchinterval of int × int × int × int
          | Kswitch of int Array.Array
```

A structured constant (*struct_constant*) is either a *string* constant, a *real* constant, an *integer* constant or a constant *block*, and a *gvar* denotes an address in the global store. A constructor tag (*constr_tag*) is basically an integer denoting the runtime tag of a constructor (see chapter 10).

The (incomplete) syntax of the primitives (the type *primitive*) is given below.

```
   datatype primitive =
            Pdummy of int
          | Pupdate | Ptag_of | Praise
          | Ptest of bool_test
          | Pfield of int | Psetfield of int
          | Pccall of string × int
          | Paddint | Pdivint
              ⋮
          | Pfloatprim of float_primitive
       and float_primitive =
            Pfloatofint
          | Pnegfloat | Paddfloat | Psubfloat | Pmulfloat | Pdivfloat
       and bool_test =
            Peq_test | Pnoteq_test
          | Pint_test of prim_test
          | Pfloat_test of prim_test
          | Pstring_test of prim_test
          | Pnoteqtag_test of constr_tag
```

A primitive test (*prim_test*) has the following form.

```
   datatype prim_test = PTeq | PTnoteq | PTlt | PTle | PTgt | PTge
```

At the time of writing the complete syntax of the sequential code corresponds closely to the syntax of the sequential code for the Caml Light system (Zam code). The link facility of the Caml Light system (see [Ler90b, chapter 6]) is not used in our implementation, hence some arguments of some of the Zam instructions (Zam instructions that have to do with global variables) are simplified. The syntax of the primitives could be arranged to fit with the *pervasives* of Standard ML.

12.2 Generation of initial code and function code

A *lambda program* is compiled into Zam code by traversal of the lambda program. When creating code for an *executor* (in contrast to creating code for an *interpreter*) [Han91], such as the Zinc abstract machine, it is necessary to generate *jumps* in the sequential code.

To compile the lambda program into sequential code the lambda program is flattened recursively. Compilation of functions and default expressions in switch constructs is delayed until the remaining lambda program has been traversed, recursively. Strictly speaking, it is not necessary to delay compilation of these lambda expressions until the whole lambda program has been traversed, but it helps the compiler generating efficient and small code since code for "jumping over" the code compiled for a function can be avoided[1]. Practically this procedure is done by use of a stack for holding delayed lambda expressions. Later, when the initial code has been generated the lambda expressions on the stack are popped and compiled, until the stack is empty. During compilation of expressions popped from the stack, compilation of new lambda expressions may be delayed and hence pushed onto the stack.

First we describe some basic functions that are needed for the compilation. We need to be able to generate a fresh label. This is done by the function *new_label*:

```
local
   val lab = ref 0
in
   fun new_label () =
            (lab := !lab + 1;
             !lab - 1)
end
```

The stack mentioned above should be global to the compilation functions. It is simply represented as a value of type *(Lambda_Exp* $\times$ *int) list ref* and is initially a reference to the empty list. There is a function *push_exp* that takes a pair of a lambda expression and a label (really an integer) as argument and returns a value of type *unit*. As a side effect the pair is pushed onto the stack. Similarly, there is a function *pop_exp* that takes a value of type *unit* as argument and returns the top element of the stack (a pair of a lambda expression and a label) while removing this element from the stack. If the stack is empty the exception *StackEmpty* will be raised.

To avoid generating code that after introducing a value in the accumulator immediately replaces it by another, the following compilation function is introduced:

[1]This technique is adopted from the Caml Light system.

```
fun into_accu v C =
    case C of
        (Kquote _ :: _) ⇒ C
      | (Kget_global _ :: _) ⇒ C
      | (Kaccess _ :: _) ⇒ C
      | (Kpushmark :: _) ⇒ C
      | _ ⇒ v @ C
```

Both arguments of the above function must be lists of Zam instructions. The second argument C is the continuation and the first argument v should be a list of instructions that introduces a value in the accumulator and besides from this has no side effects. This example shows how a continuation can be used for code optimization. We say that the accumulator is *dead*, at a given point in the Zam code if it is overwritten before it is used. This optimization assures that no value is stored in a dead accumulator. Code that introduces values in an accumulator that is dead [ASU86, page 595] is eliminated from the code. The first three cases of optimization of the continuation C in the above function are easily verified; the accumulator is overwritten by the first instruction in the continuation.

To understand that the last case of optimization of the continuation is safe, first see that a *Kpushmark* instruction will *always* (by construction) be followed by a *Kapply* instruction in the generated Zam code. In between these instructions are instructions that will introduce values in the accumulator and on the stack (by *Kpush* instructions). The value that is in the accumulator prior to execution of the *Kpushmark* instruction will not be used in between the *Kpushmark* instruction and the *Kapply* instruction (it will actually be overwritten by code following the *Kpushmark* instruction). Execution of the *Kapply* instruction causes a new value to be introduced in the accumulator and hence an instruction with no side effects, introducing a value in the accumulator prior to the *Kpushmark* instruction will have no effect. That is, the accumulator is dead if the continuation starts with a *Kpushmark* instruction, hence avoiding introducing a value in the accumulator at this place in the Zam code is safe.

The reason that this optimization works is that the value that is in the accumulator prior to execution of the *Kpushmark* instruction is not *used* in between the *Kpushmark* instruction and the *Kapply* instruction. If code sequences like

$$\ldots \mathit{Kaccess}\ 1 :: \mathit{Kpushmark} :: \mathit{Kpush} :: \mathit{Kaccess}\ 3 :: \mathit{Kapply} :: \ldots$$

could be generated by our compiler, then the optimization would be unsafe since the value that is in the accumulator prior to execution of the *Kpushmark* instruction is used in between the *Kpushmark* instruction and the *Kapply* instruction. Instead our compiler will generate code sequences like[2]:

$$\ldots \mathit{Kpushmark} :: \mathit{Kaccess}\ 1 :: \mathit{Kpush} :: \mathit{Kaccess}\ 3 :: \mathit{Kapply} :: \ldots$$

Note that this code sequence has the same *meaning* as the above code sequence.

Whenever a branch is needed in the code, the following function from the Caml Light compiler is used to avoid a jump to a jump in the code.

[2]There is one case where this is *not* the case. For compilation of a *handler* (see below) the accumulator is not dead at the point in the code where the *Kpushmark* instruction resides since the accumulator is pushed onto the stack immediately after. This is not a problem since no optimization is done on the generated code for a handler.

```
fun make_branch (C as (Kreturn :: _)) = (Kreturn, C)
  | make_branch (C as ((branch as (Kbranch _)) :: _)) = (branch, C)
  | make_branch C =
      let
        val lbl = new_label()
      in
        (Kbranch lbl, Klabel lbl :: C)
      end
```

Other optimizations (e.g. optimization on function application) needs to check whether the construction being compiled is in tail position; it is if the continuation starts with a *Kreturn* instruction. For this purpose the following test is provided.

```
fun is_return (Kreturn :: _) = true
  | is_return _ = false
```

We now show how a lambda expression is compiled into Zam code. The compilation scheme is given by the $\mathcal{C}$ compilation function. It takes as arguments a lambda expression and a *continuation* of Zam code (of type *zam_instruction list*) and returns a list of Zam code.

12.2.1 Variables and constants

A variable based on a de Bruijn index is compiled into code that accesses the value with de Bruijn index $i+1$ in the environment and puts it into the accumulator.

$$\mathcal{C}\ [\![dbVAR\ i]\!]\ C \equiv$$
$$into\_accu\ [Kaccess\ \ i]\ \ C$$

A *global* variable is compiled into code that puts the value corresponding to the *gvar* into the accumulator.

$$\mathcal{C}\ [\![dbGLOBAL\ gvar]\!]\ C \equiv$$
$$into\_accu\ [Kget\_global\ \ gvar]\ \ C$$

Compilation of an integer constant is also trivial.

$$\mathcal{C}\ [\![dbINTEGER\ n]\!]\ C \equiv$$
$$into\_accu\ [Kquote(SCatom\ \ (ACint\ \ n))]\ \ C$$

At the time of writing, string and real constants are compiled in a very inefficient way. At a later stage, string and real constants should be bound statically in the abstract machine before byte code is run. This is currently not possible since the compiler (ML Kit) and the Zinc abstract machine run concurrently in different Unix processes, communicating over two pipes on a simple protocol. It is possible however, to extend the protocol and hence obtain more efficient code. The following function introduces a string in the accumulator of the abstract machine.

```
fun comp_string s C =
  let
     fun blit_chars [] _ = []
       | blit_chars (c::rest) n =
         [Kquote(SCatom(ACint c)), Kpush,
          Kquote(SCatom(ACint n)), Kpush,
          Kaccess 0, Kprim(Psetstringchar)] @
         (blit_chars rest (n+1))
     val charlist = map (fn a ⇒ ord a) (explode s)
     val len = List.size charlist
  in
     into_accu (Kquote(SCatom(ACint len)) ::
                Kprim(Pccall("create_string", 1)) ::
                Klet :: (blit_chars charlist 0) @
                [Kaccess 0, Kendlet 1]) C
  end
```

The string is compiled into code that first allocates a string of the given size (create_string) in the abstract machine and then updates each character of the string with the correct value.

The compilation functions for string constants and real constants then become trivial.

$$\mathcal{C}\ [\![dbSTRING\ s]\!]\ C \equiv$$
$$comp\_string\ \ s\ \ C$$

To compile a real constant we first convert the real constant into a string, then change the characters that need to be changed for the C primitive *float_of_string* to work. We then generate code that introduces the string and calls the C primitive *float_of_string*.

$$\mathcal{C}\ [\![dbREAL\ r]\!]\ C \equiv$$

```
let
   val s = Real.string r
   val s1 = String.subst String.MatchCase "~" "-" s
   val s2 = String.subst String.MatchCase "E" "e" s1
in
   comp_string s2 (Kprim(Pccall("float_of_string", 1)) :: C)
end
```

The functions *List.size*, *Real.string*, *String.subst* and *String.MatchCase* are all from the Edinburgh Library [Ber91].

12.2.2 Function application

An application is compiled into code that first evaluates the argument, pushes it onto the stack, then evaluates the *function* and finally applies the function to the argument. If the application is not in tail position, code is built that first pushes a mark on the stack and then proceeds as above. The *mark* insures (in our case) that the function is applied to only *one* argument. The *Kpushmark* instruction *works* together with the *Kapply* instruction and is originally from the Caml Light system where function application on this level could be curried.

```
          $\mathcal{C}$ [[dbAPP (body, [arg])]] C ≡
(case C of
     (Kreturn :: C') ⇒
        $\mathcal{C}$ [[arg]]
        (Kpush :: ($\mathcal{C}$ [[body]] (Ktermapply :: C')))
   | _ ⇒
        Kpushmark :: ($\mathcal{C}$ [[arg]]
        (Kpush :: ($\mathcal{C}$ [[body]] (Kapply :: C)))))
```

If the application is in tail position (the continuation starts with a *Kreturn* instruction) there is no need to push a closure onto the return stack and then immediately pop it off the return stack again. To implement this optimization the Zam instruction *Ktermapply* is provided. In case of an application in tail position no mark should be pushed onto the argument stack since there will be no *Kreturn* instruction to pop it off the argument stack again.

12.2.3 Functions and let-bindings

A simple non-recursive function is compiled as follows.

```
    $\mathcal{C}$ [[dbFN body]] C ≡
if is_return C then
    Kgrab :: ($\mathcal{C}$ [[body]] C)
else
    let
        val lbl = new_label ()
    in
        push_exp (body, lbl);
        Kclosure lbl :: C
    end
```

If the abstraction is in tail position the more efficient *Kgrab* instruction is used instead of the *Kclosure* instruction (see [Ler90b, page 30]). The body of the function expression is compiled in-line instead of being pushed onto the stack of delayed lambda expressions (see below).

The compilation scheme for a *let*-construct follows.

```
       $\mathcal{C}$ [[dbLET (bind, scope)]] C ≡
let
    val C1 = if is_return C then C
                 else Kendlet 1 :: C
in
    $\mathcal{C}$ [[bind]] (Klet :: ($\mathcal{C}$ [[scope]] C1))
end
```

If the *let*-construct is in tail position there is no need to end the construct with a *Kendlet* (1) instruction since the environment will be replaced when the *Kreturn* instruction is executed.

A single recursive function binding has the following compilation scheme.

```
        $\mathcal{C}$ [[dbFIX ([(dbFN f, _)], body)]] C ≡
let
    val C1 = if is_return C then C
             else Kendlet 1 :: C
    val lbl = new_label ()
in
    push_exp (f, lbl);
    Kletrec1 lbl :: ($\mathcal{C}$ [[scope]] C1)
end
```

This is a special case of the compilation of a set of mutually recursive functions (see below). In this special case there is no need to allocate dummy variables and then later update these dummy variables. Note that since the abstraction is recursive the *name* of the function should be visible inside the body of the abstraction. This is ensured by using the *Kletrec1* instruction (see [Ler90b, page 30]). As for a *let*-construct the construct need not end with a *Kendlet1* instruction if the construct is in tail position.

To compile a set of mutually recursive function bindings all (*names* of the) functions must be visible inside every function body. We compile a set of mutually recursive function bindings into code that first introduces a *dummy* variable in the environment for each function in the set and then *updates* each entry in the environment for each corresponding function.

```
        $\mathcal{C}$ [[dbFIX (args, body)]] C ≡
let
    val s = List.size args
    val C1 = if is_return C then C
             else Kendlet s :: C
    fun comp_args _ [] = $\mathcal{C}$ [[body]] C1
      | comp_args i ((exp, sz)::rest) = $\mathcal{C}$ [[exp]]
          (Kpush :: Kaccess i :: Kprim Pupdate ::
           (comp_args (i-1) rest))
in
    List.foldR (fn (e, sz) ⇒ fn C ⇒
                Kprim(Pdummy sz) :: Klet :: C)
    (comp_args (s-1) args) args
end
```

The function *List.foldR* is from the Edinburgh Library [Ber91].

12.2.4 Exception constructs

An exception declaration is compiled using the following scheme.

$\mathcal{C}$ [[dbEXCEPTION (excon, lexp)]] C ≡

```
let
    val $C1$ = if $is\_return$ $C$ then $C$
         else $Kendlet$ 1 :: $C$
in
    $comp\_string$ ($Excon.pr\_excon$ $excon$)
    ($Kmakeblock$ ($Ref\_tag$, 1) :: $Klet$ ::
    ($\mathcal{C}$ $[\![lexp]\!]$ $C1$))
end
```

This compilation scheme ensures that the *exception name* (a reference to a string) is visible (in the *dynamic environment*) inside the scope of the exception. Code is generated that creates a string (the name of the exception constructor), builds a reference to this string (creating an *exception name*) and then binds this value in the scope of the exception declaration (*lexp*).

The compilation scheme for raising an exception is simple since the abstract machine has a primitive for this purpose.

$$\mathcal{C}\ [\![dbRAISE\ lexp]\!]\ C \equiv$$
$$\mathcal{C}\ [\![lexp]\!]\ (Kprim\ Praise :: discard\_dead\_code\ C)$$

The function *discard_dead_code* discards all code in the continuation up to the next *Klabel* instruction.

```
fun $discard\_dead\_code$ [] = []
  | $discard\_dead\_code$ ($C$ as ($Klabel$ _ :: _)) = $C$
  | $discard\_dead\_code$ (_ :: $rest$) = $discard\_dead\_code$ $rest$
```

Compilation of a handle construction follow the compilation scheme below.

```
$\mathcal{C}$ $[\![dbHANDLE\ (lexp\_body,\ lexp\_handle)]\!]$ $C \equiv$
  let
      val ($branch1$, $C1$) = $make\_branch$ $C$
      val $lbl$ = $new\_label$ ()
      val $C2$ = if $is\_return$ $C1$ then $C1$
               else $Kendlet$ 1 :: $C1$
  in
      $Kpushtrap$ $lbl$ ::
      ($\mathcal{C}$ $[\![lexp\_body]\!]$
       ($Kpoptrap$ :: $branch1$ :: $Klabel$ $lbl$ ::
       $Kpushmark$ :: $Kpush$ ::
       ($\mathcal{C}$ $[\![lexp\_handle]\!]$ ($Kapply$ :: $C2$))))
  end
```

We use the instructions *Kpushtrap*(*lbl*) and *Kpoptrap* to enclose the body of the handler. The *Kpushtrap*(*lbl*) instruction sets the current handler to *lbl*. If an exception (exception name, value) is raised by the Zam instruction *Kprim*(*Praise*) the code will continue at label *lbl* and the tuple (exception name, value) will be in the accumulator. Code is built that applies the handler (*lexp_handle*) to the tuple (exception name, value). The Zam instruction *Kpoptrap* removes the label *lbl* as the current handler and reinstalls the old handler.

12.2.5 Switch constructs

Not all switch constructs are compiled the same way. Integer switches and value constructor switches are compiled similarly since value constructors are *tagged* values and hence differentiated by integers. Because of the special nature of integers it is possible to compile integer switches and value constructor switches into very efficient code. Code can be organized as a decision tree where every node is a switch on integers which are not far apart (the distance between integers has to be less than a given constant). The compilation of a given integer or value constructor switch is done by transformation. First a decision tree is built. It is described by the datatype given below.

```
datatype decision_tree =
      DTfail
    | DTinterval of decision_tree ×
                    {low:int, act:dbLambdaExp Array.Array, high:int} ×
                    decision_tree
```

The structure *Array* is from the Edinburgh Library [Ber91]. The algorithm to construct such a balanced decision tree is fairly complex and will not be discussed here[3]. The function that implements this algorithm is *compile_nbranch*. It takes as arguments a function that maps *entries* to integers and a list of *entries* paired with lambda expressions, and it returns a decision tree (as described above). When a decision tree has been built it is possible to generate code for the switch construct. This is done (partly) by the compilation function *comp_decision_default*.

```
fun comp_decision_default tree default_lbl C =
  let
    open Array
    val (branch1, C1) = make_branch C
    fun comp_dec (DTfail) C = Kbranch default_lbl ::
            discard_dead_code C
      | comp_dec (DTinterval(left, dec, right)) C =
        let
          val (lbl_right, Cright) =
            (case right of
                 DTfail ⇒ (default_lbl, C)
               | _ ⇒ label_code (comp_dec right C))
          val (lbl_left, Cleft) =
            (case left of
                 DTfail ⇒ (default_lbl, Cright)
               | _ ⇒ label_code (comp_dec left Cright))
        in
          Kbranchinterval(#low dec, #high dec,
                          lbl_left, lbl_right) ::
            (case size (#act dec) of
               1 ⇒ $\mathcal{C}$ ⟦(#act dec) sub 0⟧ (branch1 :: Cleft)
```

[3]The algorithm is originally from the Caml Light system [Ler90b], but it has been translated into Standard ML.

```
                        | _ ⇒ comp_switch_default (#act dec)
                                    branch1 default_lbl Cleft)
            end
      in
         comp_dec tree C1
      end
```

There is a Zam instruction *Kbranchinterval*(*low, high, lbl_left, lbl_right*) that causes the control to *jump* to *lbl_left* if the integer value in the accumulator is less than the integer value *low*, to *lbl_right* if the integer value in the accumulator is greater than the integer value *right* and otherwise to the next instruction. Code for a decision tree is generated recursively by use of the function *comp_dec*. For a decision tree of the form *DTinterval*(*left, dec, right*) code is generated that given an integer value in the accumulator "jumps" to the corresponding subtree by use of the Zam instruction *Kbranchinterval*(*low, high, lbl_left, lbl_right*). If the decision tree is of the form *DTfail* code that branches to a *default* label (*default_lbl*) is generated.

The compilation function *comp_switch_default* is defined below.

```
fun comp_switch_default v branch1 default_lbl C =
   let
      open Array
      val switchtable = create (size v) 0
      fun comp_cases n =
         if n >= size v then C
         else
            let
               val lamb = v sub n
               val C' = branch1 :: comp_cases (n+1)
               val (lbl, C1) =
                  if lamb = dbSTATICFAIL then (default_lbl, C')
                  else label_code (𝒞 ⟦lamb⟧ C')
            in
               update (switchtable, n, lbl);
               C1
            end
      val code = discard_dead_code(comp_cases 0)
   in
      add_switchtable switchtable code
   end
```

This compilation function creates code for all lambda expressions in the array v and it also stores a label to each of these compiled expressions in a separate table prior to calling the *add_switchtable* compilation function. The lambda instruction *dbSTATICFAIL* is inserted in the lambda expression arrays by the *compile_nbranch* function at places where there are *holes* in the switch, and such a lambda expression compiles into a branch to the compiled code for the default expression. The *add_switchtable* compilation function is defined below.

```
fun add_switchtable switchtable C =
  let
    fun check_equal switchtab 0 = true
      | check_equal switchtab n =
          if ((switchtab sub 0) <> (switchtab sub n))
             then false
          else check_equal switchtab (n-1)
  in
    if (check_equal switchtable ((size switchtable) - 1)) then
       (case C of
           (Klabel lbl :: C1) ⇒
               if (lbl = (switchtable sub 0)) then C
               else Kbranch (switchtable sub 0) :: C
         | _ ⇒
           Kbranch (switchtable sub 0) :: C)
    else
       Kswitch switchtable :: C
  end
```

The *Kswitch* instruction takes an integer (label) array as argument. The informal semantics of this Zam instruction is to jump to the label located in the k'th cell in the array, where k is the integer value located in the accumulator. If all labels (integers) in the array (*switchtable*) are equal we simply make a branch to this label (integer), otherwise a *Kswitch* instruction is inserted in the code prior to the continuation. Note that the first part of the continuation will include code associated with the labels in the switch table (*switchtable*).

As an example we now show how an integer switch is compiled. There are two cases. One that has a default expression and one that does not. We first show how an integer switch having no default expression is compiled.

$$\mathcal{C}\ [\![dbSWITCH\_I(dbSWITCH(arg,\ casel,\ None))]\!] \quad \equiv$$

```
let
  val lbl = 0
  val C1 = comp_decision_default
              (compile_nbranch (fn i ⇒ i) casel) lbl C
in
  𝒞 ⟦arg⟧ C1
end
```

Since no jumps will be executed to the label *lbl* a dummy label is used in the compilation. First code is generated to evaluates the argument to be compared to the entries in the case list (*casel*). Then code for the compilation tree is built as described above. Note that since the case list is a list of *integers* and lambda expressions the function to pass to *compile_nbranch* is simply the identity function.

The compilation scheme for an integer switch having a default expression is given below.

$$\mathcal{C}\ [\![dbSWITCH\_I(dbSWITCH(arg,\ casel,\ Some\ lexp\_default))]\!] \quad \equiv$$

```
let
   val lbl = new_label ()
   val C1 = comp_decision_default (compile_nbranch
                  (fn i ⇒ i) casel) lbl C
in
   push_exp (lexp_default, lbl);
   $\mathcal{C}$ ⟦arg⟧ C1
end
```

Besides from delaying the compilation of the default lambda expression the compilation is as above.

Other switch constructs (real switches, string switches and exception constructor switches) are simply compiled as a sequence of branching tests. Real switches and string switches could be compiled more efficiently since an ordering relation exists. For exception constructor switches this cannot be done since no ordering exist on compile time. As an example we show how an exception switch is compiled. All exception switches in the lambda language have a default expression, hence there is only one case to consider.

$$\mathcal{C}\ [\![dbSWITCH\_E(dbSWITCH(arg,\ casel,\ Some\ lexp\_default))]\!] \ \equiv$$

```
let
   val (branch_out, C1) = make_branch C
in
   $\mathcal{C}$ ⟦arg⟧ (Kprim (Pfield 1) :: Kpush ::
            comp_exc_casel casel branch_out
               (Kpop :: ($\mathcal{C}$ ⟦lexp_default⟧ C1)))
end
```

First code is generated that evaluates the argument (an exception constructor) and then the *exception name* (a reference to a string) is extracted from the exception constructor. This is done since switches on exceptions is done on exception names. The exception case list is compiled by the compilation function *comp_exc_casel* and finally the default expression is compiled (*lexp_default*). The *comp_exc_casel* compilation function is defined below.

```
fun comp_exc_casel casel branch_out C =
   let
      fun comp [] = C
        | comp ((v, lexp) :: rest) =
         let
            val lbl = new_label ()
            val getv = case v of
                           GV gv ⇒ Kget_global gv
                         | DB i  ⇒ Kaccess i
         in
            Kpop :: Kpush :: Kpush :: getv ::
            Ktest(Pnoteq_test, lbl) :: Kpop :: $\mathcal{C}$ ⟦lexp⟧
```

```
                (branch_out :: Klabel lbl :: comp rest)
            end
    in
        comp casel
    end
```

This compilation function generates code that tests (sequentially) if the argument exception name of the switch construct is *equal* to the exception name of an entry in the exception case list. If such a test succeeds the compiled code for the corresponding lambda expression will be executed. If no test succeeds the compiled code for the default lambda expression will be executed.

12.2.6 Primitives

Most primitives are compiled trivially. In the following we show how the primitives in the lambda language, mentioned in section 11.2, are compiled into Zam code. A value constructor that takes no argument is compiled as follows.

$$\begin{gathered}\mathcal{C}\ [\![dbPRIM(dbCONprim\ i,\ [])]\!]\ C \equiv \\ into\_accu\ [Kquote(SCblock(ConstrRegular(i,\ 1),\ []))]\ C\end{gathered}$$

The compilation scheme for a value constructor applied to an argument is given below.

$$\begin{gathered}\mathcal{C}\ [\![dbPRIM(dbCONprim\ i,\ [lexp])]\!]\ C \equiv \\ \mathcal{C}\ [\![lexp]\!]\ (Kmakeblock(ConstrRegular(i,\ 1),\ 1)\ ::\ C)\end{gathered}$$

De-construction of a value constructor carrying an argument is compiled as below.

$$\begin{gathered}\mathcal{C}\ [\![dbPRIM(dbDECONprim,\ [lexp])]\!]\ C \equiv \\ \mathcal{C}\ [\![lexp]\!]\ (Kprim\ \ (Pfield\ 0)\ ::\ C)\end{gathered}$$

A constructor that takes an argument is a *block* with the argument value in the first field. To de-construct such a constructor we simply extract the argument value from this field.

As for variables there are two kinds of exception constructors in the lambda language based on de Bruijn indexes. The first kind extracts the corresponding exception name (a reference to a string) from a local variable (a de Bruijn index). The second kind extracts the corresponding exception name from a global variable (a *gvar*). The first kind takes as argument, in addition to an optional argument expression, a de Bruijn index of an exception name. An exception constructor of this kind that does not take an argument is compiled as follows.

$$\begin{gathered}\mathcal{C}\ [\![dbPRIM(dbEXCONprim\ i,\ [])]\!]\ C \equiv \\ \begin{aligned} into\_accu\ [&Kaccess\ i,\ Kpush, \\ &Kquote(SCblock(ConstrRegular(0,1),\ [])), \\ &Kmakeblock(ConstrRegular(0,0),\ 2)]\ C\end{aligned}\end{gathered}$$

The exception constructor compiles into code that constructs a block consisting of the exception name (a reference to a string) and a place holder (a *unit* value). The Zam instruction *Kquote*(*SCblock*(*ConstrRegular*(0,1), [])) simply introduces a *unit* value (of type *unit*) into the accumulator.

The compilation scheme below is for an exception constructor (of the same kind) that takes an argument.

$$\begin{array}{l}\mathcal{C}\ [\![dbPRIM(dbEXCONprim\ i,\ [lexp])]\!]\ C \equiv \\ Kaccess\ i\ ::\ Kpush\ ::\ \mathcal{C}\ [\![lexp]\!] \\ \qquad (Kmakeblock(ConstrRegular(0{,}0),\ 2)\ ::\ C)\end{array}$$

In this case the exception constructor compiles into code that constructs a block consisting of the exception name (a reference to a string) and a value.

An exception constructor that extracts the exception name from a global variable and that does not take an argument is compiled as shown below.

$$\begin{array}{l}\mathcal{C}\ [\![dbPRIM(dbGLOBALEXCONprim\ gv,\ [])]\!]\ C \equiv \\ into\_accu\ [Kget\_global\ gv,\ Kpush, \\ \qquad Kquote(SCblock(ConstrRegular(0{,}1),\ [])), \\ \qquad Kmakeblock(ConstrRegular(0{,}0),\ 2)]\ C\end{array}$$

Below is the compilation scheme for an exception constructor that takes an argument and extracts the exception name from a global variable.

$$\begin{array}{l}\mathcal{C}\ [\![dbPRIM(dbGLOBALEXCONprim\ gv,\ [lexp])]\!]\ C \equiv \\ Kget\_global\ gv\ ::\ Kpush\ ::\ \mathcal{C}\ [\![lexp]\!] \\ \qquad (Kmakeblock(ConstrRegular(0{,}0),\ 2)\ ::\ C)\end{array}$$

De-construction of an exception constructor that takes an argument is compiled by the following scheme.

$$\begin{array}{l}\mathcal{C}\ [\![dbPRIM(dbDEEXCONprim,\ [lexp])]\!]\ C \equiv \\ \quad \mathcal{C}\ [\![lexp]\!]\ (Kprim\ (Pfield\ 0)\ ::\ C)\end{array}$$

The field holding the argument value is simply extracted from the exception constructor (a block of *size* two).

Empty records are compiled using the scheme below.

$$\begin{array}{l}\mathcal{C}\ [\![dbPRIM(dbRECORDprim,\ [])]\!]\ C \equiv \\ into\_accu\ [Kquote(SCblock(ConstrRegular(0{,}1),\ []))]\ C\end{array}$$

Nonempty records are compiled as follows.

$$\mathcal{C}\ [\![dbPRIM(dbRECORDprim,\ lexps)]\!]\ C \equiv$$

```
let
    fun compexp_list [] C = C
      | compexp_list [lexp] C = $\mathcal{C}$ [[lexp]] C
      | compexp_list (lexp::rest) C =
            compexp_list rest (Kpush :: ($\mathcal{C}$ [[lexp]] C))
in
    compexp_list lexps (Kmakeblock ((ConstrRegular(0,0)),
            List.size lexps) :: C)
end
```

First the values of all components, except the first one, are computed and pushed onto the stack (note the reverse order). The first one is evaluated and put in the accumulator, then the *Kmakeblock* instruction pops all arguments from the stack and creates a *record* containing the values of the arguments.

Compilation of selection of a value from a record is as follows.

$$\mathcal{C}\ [\![dbPRIM(dbSELECTprim\ n,\ [lexp])]\!]\ C \equiv$$
$$\mathcal{C}\ [\![lexp]\!]\ (Kprim\ (Pfield\ n)\ ::\ C)$$

The pervasives are compiled trivially. We show below how the construct for integer addition is compiled.

$$\mathcal{C}\ [\![dbPRIM(dbPLUS\_INTprim,\ [lexp1,\ lexp2])]\!]\ C \equiv$$
$$\mathcal{C}\ [\![lexp2]\!]\ (Kpush\ ::\ (\mathcal{C}\ [\![lexp2]\!]\ (Kprim\ Paddint\ ::\ C)))$$

For pervasives like **div** that may raise an exception the corresponding exception constructor is passed as an extra argument to the primitive. When compiling such pervasives code is generated such that the primitive in the abstract machine can raise the exception if needed. How the construct for integer division is compiled is shown below.

$$\mathcal{C}\ [\![dbPRIM(dbDIV\_INTprim,\ [lexp1,\ lexp2,\ lexp3])]\!]\ C \equiv$$
$$\mathcal{C}\ [\![lexp2]\!]\ (Kpush\ ::\ (\mathcal{C}\ [\![lexp2]\!]\ (Kpush\ ::\ \mathcal{C}\ [\![lexp3]\!]\ (Kprim\ Pdivint\ ::\ C)))$$

12.3 Generating binding code

It is the purpose of the binding code to bind values and exception names[4] to global variables (*gvars*). Only variables and exception names that should be visible on top level (their scope is the rest of the session) should be bound. The names of these lambda variables and exception constructors are found statically and are hence available for use in the compilation. The compiler contains a *dynamic* environment mapping lambda variables (*lvars*) to *gvars* and names of long exception constructors (*longexcons*) to *gvars* (see section 11.3 for a description).

[4]Only the exception names (references to strings), not the entire exception constructors, are bound. Similarly, only exception names are bound to de Bruijn indexes.

When the *initial* code for a top level declaration has been executed there is a record located in the accumulator. This record includes all values and exception names, introduced by the top level declaration that should be visible on top level. The binding code simply extracts the values and exception names from the record and *stores* the values and the exception names in the global variables to which they are associated in the dynamic environment.

Chapter 13

Code for the Abstract Machine

The symbolic sequential code (the Zam code) need to be translated into numeric byte code (Zinc code) for the Zinc abstract machine to run it. Most of the Zam instructions correspond directly to one byte of Zinc code. However, there are some Zinc instructions that take arguments. For these instructions the Zinc abstract machine interprets the next byte(s) in the code as argument(s) and not as instruction(s). An argument of a Zinc instruction can either be of size one *byte*, two bytes (a *short*) or four bytes (a *word*). The instructions are listed in appendix C where each instruction (byte) is given a symbolic name.

Not all instructions available in the abstract machine are used and not all pervasives of Standard ML are translated into Zinc instructions at the time of writing. In the following we use the letter $\mathcal{Z}$ to denote a translation from Zam code (a list of Zam instructions) to Zinc code (a string of bytes).

The implementation of the $\mathcal{Z}$ translation step in the new back-end of the ML Kit system uses a buffer (a byte array) to obtain higher efficiency instead of the concatenation operator (ˆ) used below. For a discussion of the abstract machine and its limitations see chapter 10.

13.1 Primitive output functions

The compiler includes a few primitive output functions which are used in the translation function (the $\mathcal{Z}$ scheme). The function *out: int → string* converts an integer to a string of length one, holding the integer as a character. This function is used for converting names of Zinc instructions (really integers) and bytes of integer arguments to characters (strings of length one). Similarly the functions *out_short* and *out_long* convert integer arguments to strings of length two and four, respectively (most significant byte first).

To introduce integer constants in the abstract machine the function *out_int_const* is provided.

```
    fun out_int_const i =
        if (i <= 127 andalso i >= 0) then
            (out CONSTBYTE) ^ (out (i+i+1))
    else
      if ($i$ <= 16383 andalso $i$ >= ~16384) then
        ($out\ CONSTSHORT$) ^ ($out\_short\ (i{+}i{+}1)$)
      else
      $Crash.unimplemented$ "$out\_int\_const : out\ of\ range$"
end
```

At the time of writing integer constants larger than 16383 or lower than ∼ 16384 cannot be introduced in the Zinc abstract machine[1].

13.2 The $\mathcal{Z}$ translation scheme

Most Zam instructions are translated trivially, though there are some *nontrivial* (read: not *so* trivial) translation steps. Because of the similarity of most of the translations only a few examples will be given.

As an example of a trivially translated instruction we show below how the Zam instruction *Kprim*(*Paddint*) is translated into Zinc code.

$$\mathcal{Z}\ [\![Kprim(Paddint) :: C]\!]\ \equiv\ (out\ ADDINT)\ \hat{}\ (\mathcal{Z}\ [\![C]\!]\ )$$

All Zam instructions taking arguments are translated into Zinc instructions taking arguments. We show below how the Zam instruction *Kget_global* (*gv*) is translated.

$$\mathcal{Z}\ [\![Kget\_global\ gv :: C]\!]\ \equiv\ (out\ GETGLOBAL)\ \hat{}\ (out\_short\ gv)\ \hat{}\ (\mathcal{Z}\ [\![C]\!]\ )$$

To translate the introduction of an integer value into the accumulator the following translation scheme is given.

$$\mathcal{Z}\ [\![Kquote(SCatom(ACint\ i)) :: C]\!]\ \equiv\ (out\_int\_const\ i)\ \hat{}\ (\mathcal{Z}\ [\![C]\!]\ )$$

Jumps in the abstract machine are relative jumps. Translation of a *Klabel*(*lbl*) Zam instruction generates no Zinc code. The function *define_label* enters the label (*lbl*) and the code pointer (the index in the Zinc code string) in a table for use when a *branch* instruction is translated.

$$\mathcal{Z}\ [\![Klabel\ lbl :: C]\!]\ \equiv$$

```
let
  val _ = $define\_label\ lbl$
in
  $\mathcal{Z}\ [\![C]\!]$
end
```

[1]In the Caml Light system *larger* integer constants are stored in global variables (statically) at link time prior to execution of the code.

The function *define_label* also takes care of *back-patching* (updating) previously translated forwarding branches to the label *lbl*. This can be done since if a label *lbl* is *undefined* when translating a *branch* to this label the branch must be a forward branch. In this case a *dummy* label (a *short*) is *emitted* and the code pointer at this place of the Zinc code is appended to a list associated with the label *lbl*. This is done by the function *out_label*. The function *define_label* updates (back-patches) all points in the code pointed to by the members of the list associated to the label *lbl* with relative addresses to the point in the code associated with the label. As an example of how a branch is translated we show below how the Zam instruction *Kbranch* (*lbl*) is translated into Zinc code.

$$\mathcal{Z}\ [\![Kbranch\ lbl :: C]\!] \ \equiv\ (out\ BRANCH)\ \hat{}\ (out\_label\ lbl)\ \hat{}\ (\mathcal{Z}\ [\![C]\!]\ )$$

The translation scheme of the Zam instruction *Kbranchinterval*(*low, high, lbl_low, lbl_high*) is given below.

$$\begin{aligned}&\mathcal{Z}\ [\![Kbranchinterval(low, high, lbl\_low, lbl\_high) :: C]\!] \ \equiv \\ &\quad (out\ PUSH)\ \hat{}\ (out\_int\_const\ low)\ \hat{}\ (out\ PUSH)\ \hat{} \\ &\quad (\textbf{if}\ low <> high\ \textbf{then}\ out\_int\_const\ high\ \textbf{else}\ "")\ \hat{} \\ &\quad (out\ BRANCHINTERVAL)\ \hat{}\ (out\_label\ lbl\_low)\ \hat{} \\ &\quad (out\_label\ lbl\_high)\ \hat{}\ (\mathcal{Z}\ [\![C]\!]\ )\end{aligned}$$

The generated code pushes the test value onto the stack, introduces the *low* and *high* integer values (the *high* integer value only if necessary) and finally executes the Zinc instruction *BRANCHINTERVAL* which takes two arguments. The first argument of the *BRANCHINTERVAL* instruction is the address in the generated code to jump to if the test value is lower than the value *low*, whereas the second argument is the address to jump to if the test value is higher than the value *high*. If the test value is between the two values the code following the *BRANCHINTERVAL* instruction is executed.

Chapter 14

Value Printing

In the new back-end of the ML Kit system values are computed and stored in the global store in the Zinc abstract machine. For the user to *see* these values we must generate code to print every value resulting from evaluation. We generate Zinc code that prints the value located in the given index of the global store. Given the type of the value, Zam code for printing can be generated and then translated to Zinc code (see chapter 13).

There are two ways in which the type of a value in the ML Kit system can be extracted. One way is to extract the type directly from the *static environment* of the ML Kit system [BRTT93, page 50]. Another way is to extract the type from the typed lambda language. The first way gives us the advantage of being able to print records with labels. This is not possible if the second way is chosen since the typed lambda language has no notion of labels. However, if instead we chose to extract the type of a value from the static environment of the ML Kit system, it is not possible to print arguments to constructors. Unfortunately, it is *not* possible to join the two methods since we need to extract subtypes recursively from a given type.

At the time of writing the value printer extracts the type of a given value from the static environment of the ML Kit system. Therefore, it is not currently possible to print arguments to constructors. In the future the value printer should extract the type of a given value from the typed lambda language. If this approach is used, labels must be associated to each *field* of a record type in the typed lambda language to print records with labels.

First we describe how a type is extracted from the static environment and also how subtypes of this type are extracted. We then show how Zam code (which is translated into Zinc code and run) is constructed to print a value of a given type.

14.1 Types of values to print

The main printing function is passed a tag environment (see chapter 9), a global variable (an index to a global store), and a *typescheme* from which a *Type* can be extracted. The structure *StatObject* that gives access to the type *Type* and the *extraction functions* matches a signature

STATOBJECT[1]. The part of the signature that is important for value printing is shown below.

```
signature STATOBJECT =
  sig
    ⋮
    type TypeScheme
    val unTypeScheme : TypeScheme → TyVar list × Type

    type Type and FunType and ConsType and RecType
    val equal_Type : Type × Type → bool

    val unTypeRecType : Type → RecType Option
    val unTypeFunType : Type → FunType Option
    val unTypeConsType : Type → ConsType Option

    val unRecTypeSorted : RecType → (lab × Type) list
    val unConsType : ConsType → (Type list × TyName) Option

    val TypeUnit : Type
    val TypeInt : Type
    and TypeReal : Type
    ⋮
  end
```

The types *TyVar*, *TyName* and *lab* are the types for a type variable, a name of a type and a label. The *type extraction functions* mentioned in the signature allow us to traverse a type recursively until we reaches a basic type (*TypeUnit*, *TypeInt* or *TypeReal*), an empty record type or a constructed type. When reaching a constructed type we generate code that checks what constructor to print. This is done by looking up constructor tags (integers) in the tag environment (see chapter 9). To get the constructors (*longcons*) for a given type name (*TyName*) these can also be looked up in the tag environment. The tag environment has been added to the dynamic basis of the compiler (see [BRTT93, page 67]).

As mentioned above it is not possible to generate code that prints the arguments of a constructor since the type of such an argument cannot be extracted. The *Type list* part of the optional result returned by the function *unConsType*, mentioned in the signature above, only contains the types *instantiated* for the type variables of the type scheme inferred for the original datatype declaration. If there were an environment mapping constructors (*longcons*) to *optional* types (the type of the optional argument of a constructor) it would be possible to *substitute* the type variables in such types with the types given in the *Type list* part of the optional result returned by the function *unConsType* and hence get the type of the argument of a given constructor.

14.2 Generating printing code

First we describe some basic code sequences for printing. These basic code sequences are put together and integrated with a set of mutually recursive functions for generating code for

[1]The structure *StatObject* and the signature *STATOBJECT* are parts of the ML Kit system.

printing values of different types. There is a main function that surrounds the set of mutually recursive functions. The purpose of this function is to generate code that extracts the value to print from the global store. The printing code that is generated will then, by use of the stack, traverse the value and use specific primitives of the Zinc (Zam) abstract machine to print sub-values of different types.

14.2.1 Auxiliary functions

All basic printing primitives write their output to an *output channel*. This channel is bound in the environment for the entire printing code by initial code generated by the following function.

```
fun code_get_gv gv =
      [Kquote (SCatom (ACint 1)),
       Kprim (Pccall ("open_descriptor",1)),
       Klet, Kget_global gv]
```

The generated code first introduces the integer value one, indicating standard output (*std_out*), and then calls the primitive *open_descriptor* that returns a *channel* on which *output* from code generated by printing functions can be written. This channel is bound in the current environment and the value to print (located in the global store) is initially put in the accumulator.

To flush the output of the Zinc abstract machine the following code sequence is used.

```
val code_flush =
      [Kaccess 0, Kprim (Pccall ("flush",1)), Kendlet 1]
```

This code sequence should only be appended at the end of the printing code since the *Kendlet* 1 instruction will remove the environment containing the *output channel*.

14.2.2 Printing base values

For each of the code sequences below the value to print is pushed onto the stack by the Zam instruction *Kpush*. Then the *output channel* is accessed in the local environment and finally the necessary output primitive (C primitive) is called.

To print an *integer* value, a *real* value, or a *string* value (located in the accumulator), the following code sequences are used.

```
val code_print_int =
      [Kpush, Kaccess 0, Kprim (Pccall ("output_int_val", 2))]
val code_print_real =
      [Kpush, Kaccess 0, Kprim (Pccall ("output_float_val", 2))]
val code_print_string =
      [Kpush, Kaccess 0, Kprim (Pccall ("output_string_val", 2))]
```

To print an *exception constructor* the exception name to print is first extracted from a block containing as a component a reference to a string (also a block).

```
val code_print_exn =
        [Kprim(Pfield 1), Kprim(Pfield 0), Kpush, Kaccess 0,
         Kprim (Pccall ("output_string_val", 2))]
```

When generating printing code it is necessary to generate code to print characters and strings known only to the ML Kit system. These strings could be labels, parentheses etc. To generate code that prints such strings the code generating function *code_string* is provided.

```
fun code_string s =
   let
      val l = explode s
      fun code_char c =
            [Kquote(SCatom (ACint (ord c))), Kpush, Kaccess 0,
             Kprim (Pccall ("output_char", 2))]
      fun code_string_list [] = []
        | code_string_list (a::r) =
              code_char a @ code_string_list r
   in
      [Kpush] @ (code_string_list l) @ [Kpop]
   end
```

The code generated by this function first pushes the value in the accumulator onto the stack for later use. Then code is generated that will print each character in the string one at a time and finally the value in the accumulator is restored (popped from the stack).

14.2.3 Printing structured values

The code generating functions for printing structured values are described below. These functions use the type extracting functions described in section 14.1 to determine what kind of code should be generated. There is a function that generates code to print a value of any given type. This function then calls appropriate functions to generate code that prints records, constructors, etc. The function *code_print_val* takes as argument a type and generates code to print a corresponding value of that type.

```
fun code_print_val typ =
   if equal_Type (typ,TypeInt) then code_print_int
   else
      if equal_Type (typ,TypeReal) then code_print_real
      else
       (case unTypeFunType typ of
            Some _ => (code_string "fn")
          | None =>
```

```
            (case unTypeRecType typ of
                Some rectyp ⇒
                 code_print_rec (unRecTypeSorted rectyp)
              | None ⇒
                 (case unTypeConsType typ of
                     Some constyp ⇒
                      (case unConsType constyp of
                          Some (types, tyname) ⇒
                           (case tyname of
                                TyName.tyName_STRING ⇒
                                     code_string "\"" @
                                     code_print_string @
                                     code_string "\""
                              | TyName.tyName_REF ⇒
                                     code_string "ref"
                              | TyName.tyName_EXN ⇒
                                     code_print_exn @
                                     code_string "(-)"
                              | _ ⇒ code_print_con tyname
                           )
                        | None ⇒ code_string "#")
                   | None ⇒ code_string "#"))
        )
```

The function "branches out" on the given type and calls the corresponding function for generating printing code. For functional values code is generated that prints the string "fn", and for record values (and tuples) the function *code_print_rec* is called. To check whether a given constructed type is a string, a reference value or an exception constructor, the type names *TyName.tyName_STRING*, *TyName.tyName_REF* and *TyName.tyName_EXN* are tested against the type name of the constructed type. To generate code that prints the name of a value constructor the function *code_print_con* is called.

The code generating function used to generate code for printing records and tuples is shown below.

```
and code_print_rec rectyp =
  let
    fun is_rec_tuple rectyp =
      let
        fun tupleness [] _ = true
          | tupleness ((lab, _) :: r) n =
              if is_LabN (lab, n) then tupleness r (n + 1)
              else false
      in
        case rectyp of
            [_] ⇒ false
          | _ ⇒ tupleness rectyp 1
      end
    fun code_print_tup [] _ = []
      | code_print_tup [(_, typ)] n =
          [Kprim (Pfield n)] @ (code_print_val typ)
      | code_print_tup ((_, typ) :: p2 :: rest) n =
          [Kpush, Kprim (Pfield n)] @ (code_print_val typ) @
          (code_string ",") @ [Kpop] @
          (code_print_tup (p2 :: rest) (n + 1))
    fun code_print_rec2 [] _ = []
      | code_print_rec2 [(lab, typ)] n =
          (code_string (pr_Lab lab ^ "=")) @
          [Kprim (Pfield n)] @ (code_print_val typ)
      | code_print_rec2 ((lab, typ) :: p2 :: rest) n =
          (code_string (pr_Lab lab ^ "=")) @
          [Kpush, Kprim (Pfield n)] @ (code_print_val typ) @
          (code_string ",") @ [Kpop] @
          (code_print_rec2 (p2 :: rest) (n + 1))
  in
    if is_rec_tuple rectyp then
       ((code_string "(") @ (code_print_tup rectyp 0) @
        (code_string ")"))
    else
       ((code_string "{") @ (code_print_rec2 rectyp 0) @
        (code_string "}"))
  end
```

If the value to be printed is determined to be a *tuple* (each label is equal to the index in the record, starting with 1 and the number of values in the record does not equal 2) code is generated that prints the value on the form

$$(v_1,\ v_2, \cdots,\ v_n)$$

where n equals the number of values in the record (tuple). Otherwise code is generated that prints the value on the form

$$\{l_1 = v_1,\ l_2 = v_2, \cdots,\ l_n = v_n\}$$

where n equals the number of values in the record and where $l_k,\ k \in \{1,\ 2, \ldots,\ n\}$ is the k'th label of the record.

To generate code that prints the name of a constructor the following function is provided.

```
and code_print_con tyname =
  let
    val cons = (TagEnv.lookupCons tagenv tyname
                handle TagEnv.LookUp => Crash.impossible
                                        "code_print_con.cons")
    val tags = (map (fn con => TagEnv.lookupTag tagenv con) cons
                handle TagEnv.LookUp => Crash.unimplemented
                                        "code_print_con.tags")
```

```
        val (lbl_end, C) = label_code [Kpop]
        val v = Array.create (List.size tags) 0
        fun gen_switch_code [] [] = []
          | gen_switch_code (con :: cons') (tag :: tags') =
              let
                 val (lbl, C1) =
                    label_code (code_string (Con.pr_con con))
                 val _ = Array.update (v, tag, lbl)
              in
                 C1 @ [Kbranch lbl_end] @
                 (gen_switch_code cons' tags')
              end
          | gen_switch_code _ _ = Crash.impossible "code_print_con"
    in
        [Kpush, Kprim(Ptag_of), Kswitch v, Kbranch lbl_end] @
        (gen_switch_code cons tags) @ C
    end
```

When generating code for printing the name of a constructor the list of constructors (*longcons*) for the given constructed type is first looked up in the tag environment. Then the tag for each constructor is looked up in the same environment. The constructed code is simply a *switch* on the tags of the constructors. Code associated with each branch in the switch prints the name of the corresponding constructor.

At the time of writing it is not possible to print arguments to constructors, and especially arguments to the reference constructor *ref* cannot be printed. At a later stage however, when this is possible, we must check for cycles in data structures containing *ref*-constructions in substructures, when generating printing code. This is necessary to ensure that execution of the generated printing code terminates. Another (and easier) approach is to generate code that only prints a given number of *levels* of a data structure.

Chapter 15

The Module Language

The module language of Standard ML has a complicated static semantics. The static semantics of Standard ML corresponds to the *elaboration* part of the ML Kit system, for which phrases of the module system succeed to elaborate. The dynamic semantics for the module language [MTH90, chapter 7] is relatively simple. At present the ML Kit system does not compile phrases of the Standard ML module language into the typed lambda language described in section 11. Literature regarding implementation of the Standard ML module system includes [App92, AM87, AM91, Mac88].

The declaration compiler of the core language in the ML Kit system is implemented in *monadic* style [Wad92]. A similar technique can be used in an implementation of the top-level declaration (*topdec*) compiler.

In this chapter we first discuss how constructs of the module language can be compiled into an untyped lambda language. We then proceed to suggest how the compilation step may be extended such that a *type* for each construct in the lambda language can be determined.

15.1 Compilation of the module language

As mentioned above, the dynamic semantics of the module language of Standard ML is described in [MTH90, chapter 7]. To implement the compiler *correctly* we need to generate code that *operates* according to this dynamic semantics. There are a few errors in the dynamic semantics described in [MTH90]. Datatype specifications cannot be omitted from the dynamic semantics of the module language [Kah93, page 26] as suggested in [MTH90, chapter 7].

Signatures in the dynamic semantics evaluate to interfaces. However, as mentioned in [MTH90, page 58] interfaces are naturally obtained from the static elaboration[1]. Hence, only the rules 160–169, 187–191 and 193 of [MTH90] may cause code to be introduced.

We first discuss how to represent different *objects* of the module system at runtime and then discuss the *operations* that are necessary on these objects.

[1] *Signature* rules are included in the dynamic semantics, in [MTH90], to separate the dynamic and static semantics for presentation reasons.

15.1.1 Representation

We need no notion of *signatures* (or interfaces) at runtime since the required information can be obtained from the static elaboration. A *structure* in the module system is represented as a frame. A *frame* is a collection of ML values and exception names *listed* in an order statically inferred from the signature (interface) of the corresponding structure. In an untyped lambda language a *frame* could be represented as a tuple (record).

A *functor* of the Standard ML module language is represented as a function taking as argument a frame and returning as a result a new frame.

15.1.2 Operations

Several operations regarding the module system need to be defined [MTH90, chapter 7]. In this section we use an untyped form of the typed lambda language of chapter 11 to sketch what kind of code is generated for each operation. The derived forms [MTH90, page 68] are not considered.

Apart from the sub-environments of the compile time environment of the core declaration compiler, there is a need for a sub-environment mapping structure identifiers (*strids*) to lambda variables (*lvars*) and a sub-environment mapping functor identifiers (*funids*) to lambda variables (*lvars*).

Compiling structure expressions (*strexp*)

To compile a structure expression of the form

struct *strdec* **end**

we first compile the structure-level declaration *strdec* [MTH90, rule 160]. This results in a list of lambda expressions *lexps* (a structure-level declaration can be a sequence of structure-level declarations [MTH90, rule 168]). We then build a record (tuple) in the lambda language. The generated code for this operation is as follows.

$$PRIM(RECORDprim,\ lexps)$$

For a structure expression of the form

longstrid

we need to look up the lambda variable (*lvar*) associated with the long structure identifier (*longstrid*) [MTH90, rule 161]. If the identifier is simple (if it is not enclosed by a structure) the construct compiles into

$$VAR(lvar)$$

Otherwise, if the identifier is not simple we need to extract the value (sub-frame) from the frame, recursively. The structure expression compiles into

$$PRIM(SELECTprim\ a_1,\ [\ldots,\ [PRIM(SELECTprim\ a_n,\ [VAR\ lvar])]\ \ldots\ ])$$

where $a_k,\ k > 0$ is the index for the k'th *structure* of the long structure identifier (*longstrid*).

For a structure expression of the form

$$funid\ (\ strexp\ )$$

we need to generate code that *applies* the functor (*funid*) to the frame (*strid*) [MTH90, rule 162].

First we need to define a *trimming* operation that at runtime *trims* a frame and as a result produces a new frame. A trimming operation is a lambda construct that is produced given an interface for the argument frame and an interface for the resulting frame. The generated lambda construct must, given an argument frame, create a new frame lay-outed as required by the result interface. This operation includes cut-down and reordering of the argument frame by selecting and reordering components of the argument frame. And if an identifier, say A, is a constructor in the argument interface and a value in the result interface, then a new corresponding field must occur in the result frame, roughly corresponding to the binding "**val** $A = S.A$", where S is the (structure) identifier for the argument frame. Sub-frames (substructures) of the frame to be trimmed must be trimmed, recursively, by the constructed code.

The functor application should generate code that first evaluates the structure expression *strexp*, then trims the resulting frame to suit the functor argument signature, and then applies the functor *funid* to this trimmed frame. The result is a new frame. We need to look up the lambda variable (*lvar*) associated with the functor identifier *funid*. The generated code for this operation is as follows.

$$APP(VAR\ lvar,\ trimmed\_frame)$$

where *trimmed_frame* is code for trimming the frame associated with the structure expression *strexp*.

Structure expressions of the form

$$\textbf{let}\ strdec\ \textbf{in}\ strexp\ \textbf{end}$$

are compiled as the corresponding expression of the core language [MTH90, rule 163]. The generated code must first evaluate the structure-level declaration *strdec* (see below), bind it in the enclosing environment and then evaluate the structure expression *strexp*.

Compiling structure-level declarations (*strdec*)

A structure-level declaration of the form

$$dec$$

should cause code for the core declaration *dec* to be generated.

For a structure-level declaration of the form

$$\textbf{structure}\ strbind$$

where *strbind* is of the form

$$strid\ \langle :\ sigexp \rangle\ =\ strexp\ \langle\langle\ \textbf{and}\ strbind\ \rangle\rangle$$

we need to generate code that for each *structure binding* (separated by **and**) binds the resulting frame in the enclosing environment [MTH90, rule 165, 169]. For each structure binding *strbind*, we first generate code that evaluates the structure expression *strexp*. If the structure identifier *strid* is constrained by a signature expression (an interface), the frame is trimmed (see above) to suit the interface.

Structure-level declarations of the form

$$\textbf{local}\ \textit{strdec}_1\ \textbf{in}\ \textit{strdec}_2\ \textbf{end}$$

are compiled as the corresponding declaration of the core language [MTH90, rule 166]. The generated code must first evaluate the first structure-level declaration $\textit{strdec}_1$ (see below), bind it in the enclosing environment and then evaluate the second structure-level declaration $\textit{strdec}_2$.

The *empty* structure-level declaration [MTH90, rule 167] causes, as the empty declaration of the core language, no code to be generated.

A structure-level declaration of the form

$$\textit{strdec}_1\ \langle\ ;\ \rangle\ \textit{strdec}_2$$

is compiled as the corresponding declaration of the core language [MTH90, rule 168]. The generated code evaluates the first structure-level declaration $\textit{strdec}_1$ and then evaluates the second structure-level declaration $\textit{strdec}_2$.

Compiling functor declarations (*fundec*)

A functor declaration of the form

$$\textbf{functor}\ \textit{funbind}$$

where *funbind* is of the form

$$\textit{funid}\ (\ \textit{strid}\ :\ \textit{sigexp}\ )\ \langle :\ \textit{sigexp'} \rangle\ =\ \textit{strexp}\ \langle\langle\ \textbf{and}\ \textit{funbind}\ \rangle\rangle$$

is compiled into a closure (a function closure can be used in an implementation) that when applied to an argument evaluates the structure expression *strexp* in the environment contained in the closure [MTH90, rule 187, 188]. Compilation of the functor body *strexp* can assume that the argument has the form prescribed by the argument interface inferred for the signature expression *sigexp*. If the declaration is constrained by a signature expression *sigexp'* the generated code should apply the trimming operation to the resulting frame.

No code is generated for an empty functor declaration [MTH90, rule 189].

A functor declaration of the form

$$\textit{fundec}_1\ \langle\ ;\ \rangle\ \textit{fundec}_2$$

is compiled as the corresponding declaration of the core language [MTH90, rule 190]. The generated code evaluates the first functor declaration $\textit{fundec}_1$ and then evaluates the second functor declaration $\textit{fundec}_2$.

Compiling top-level declarations (*topdec*)

For each top-level declaration code for the corresponding *declaration* is generated [MTH90, rule 191, 193]. No code is generated for a signature declaration *sigdec*.

Compilation of the open declaration of the core language (*dec*)

When extending the core language of Standard ML with the module language an **open** declaration is introduced in the core language. For a core declaration of the form

$$\textbf{open}\ \mathit{longstrid}_1\ \cdots\ \mathit{longstrid}_n$$

where $n \geq 1$, we need to bind all components mentioned in the interfaces for the long structure identifiers $\mathit{longstrid}_1\ \cdots\ \mathit{longstrid}_n$ in the enclosing environment [MTH90, rule 132].

Compilation of atomic expressions of the core language (*atexp*)

When extending the core language of Standard ML with the module language, atomic expressions of the core language need to be changed slightly. Atomic expressions of the form

$$\mathit{longvar}$$

are split into two cases [MTH90, rule 104]. If the long variable name is simple (if it has no structure name prefixes) then the atomic expression is compiled as usual. Otherwise, we need to extract the variable recursively from the corresponding frames.

For atomic expressions of the form

$$\mathit{longexcon}$$

a similar approach is used [MTH90, rule 106]. If the long exception constructor is simple (if it has no structure name prefixes) then the atomic expression is compiled as usual. Otherwise, we need to extract the exception name *en* recursively from the corresponding frames.

15.2 Typing the constructs in the lambda language

As described above a *frame* can be represented as a tuple (record) in the untyped lambda language. To be able to type all constructs of this lambda language and to keep a simple correspondence between structures and tuples and between functors and functions, we *must* choose how to represent exception names, since exception names may become fields in a tuple in our scheme. Fortunately, there is a simple way of representing exception names. Representing exception names as references to strings has many advantages [App92], and this representation is chosen by most Standard ML implementations. It is worth noticing that when choosing a representation for exception names (e.g. *string ref*), frames become records also in a scheme compiling abstract syntax constructs into the typed lambda language. Choosing representation however, may restrict what can be said about the translation. And certainly, if choosing representation, we loose abstraction.

A typed lambda *program* of the ML Kit system supplies the lambda language construct with a list of mutually recursive datatype bindings such that constructor names and types of arguments to constructors can be extracted. This scheme may have to be changed in some way when implementing the top-level declaration compiler. At the time of writing it is not clear what changes are necessary.

Chapter 16

Conclusion

The goal was to implement a portable Standard ML compiler that generates compact code. The first approach to this goal was to change the front-end of an existing compiler into a Standard ML compiler. The result of this work is a compiler that compiles a large subset of the core Standard ML language and that has its own module system that supports separate compilation. This compiler is capable of compiling many core Standard ML programs, though it lacks features such as overloading.

During this work many aspects of implementation of a core Standard ML compiler have been investigated. These aspects include static aspects such as lexical analysis, parsing, error handling, infix resolution and type checking, but also dynamic aspects such as order of evaluation and semantics of primitives such as equality.

However, to achieve a full Standard ML compiler this way, it would be necessary to implement a type checker completely from scratch and also to implement elaboration of modules. All this work was already done for the ML Kit system.

The second approach to the above goal was to write a new back-end for the ML Kit system[1]. The ML Kit system is very modular and it is relatively easy to replace parts of the system with new parts (chapter 9). At the time of writing the ML Kit system elaborates *all* phrases (including phrases of the module language) of Standard ML, and it compiles phrases of core Standard ML into a typed lambda language.

The new back-end translates phrases of the typed lambda language of the ML Kit system into phrases of a simpler lambda language based on de Bruijn indexes (chapter 11). Such phrases are then compiled into symbolic sequential code (chapter 12) which is translated into a string of byte code (chapter 13). This string of byte code is then executed on the modified Zinc abstract machine (chapter 10) and for each value to be printed byte code is generated and executed (chapter 14).

Though compilation of some of the lambda constructs need to be optimized the generated code for the ML Kit system integrated with the new back-end is rather small. As an example one may notice that a naïve Fibonacci function compiles into a string of less than 60 bytes.

[1]The version of the ML Kit system that has been used is the 1.0 version with a few extensions (as of April 6, 1994). The lambda language is now a typed language and core elaboration is more efficient.

Many dynamic aspects of an implementation of a Standard ML compiler have been investigated during this work. These aspects include representation of data in memory, dynamic semantics of phrases of Standard ML, integration of the compiler and the runtime system, and value printing.

We have suggested, in chapter 15, how the ML Kit system could be extended such that the compiler also compiles phrases of the module language into the typed lambda language. This extension is necessary, among other efficiency optimizations, to *bootstrap* the ML Kit system and hence achieve a portable Standard ML compiler that generates small efficient code.

Having reached the end of this work, the least I can say is that I have learned a lot about the semantics of Standard ML and that implementation of a Standard ML compiler is not at all a trivial task.

Further work

Lots of work need to be done before a full bootstrapped Standard ML compiler building on the ML Kit system is available for general use. First of all we need to implement the suggested extension to the compiler, such that phrases of the module language of Standard ML compiles into phrases of the typed lambda language (chapter 15). It is also necessary to optimize the elaboration of modules in the ML Kit system, since naïve implementations of some algorithms for the elaboration of modules cause the compiler to be practically unusable for large code segments.

At the time of writing the ML Kit system and the abstract machine runs concurrently on two Unix processes, communicating over two pipes (section 10.2). At some point (when bootstrapping the system) it must be possible to execute code located in a file, and it must be possible from *within the byte code* to execute *separate sequences of byte code* (possibly located in a separate file) on the abstract machine. This is the *key* to bootstrapping.

Apart from these needs it is also necessary to implement the complete set of primitives of Standard ML including primitives for input/output (streams). Implementation of these primitives is a fairly trivial task. Partly because many of the primitives are very simple and partly because corresponding primitives are already parts of the Zinc abstract machine.

The Zinc abstract machine of the Caml Light system has a few limitations (section 10.3). It is necessary to eliminate most of these limitations of the Zinc abstract machine for bootstrapping to be possible. Fortunately, as we have seen in section 10.3, these limitations can relatively easily be eliminated.

Appendix A

Executable and Source Code for Mini ML

This appendix includes a description of how to test the $\mathcal{M}ini\mathcal{M}l$ system and a list of all source code files that have been constructed or altered.

Execution file

To start the $\mathcal{M}ini\mathcal{M}l$ system you should be able to access the directory */home/mael/bin* located on *idfs4* at the Technical University of Denmark. To start the $\mathcal{M}ini\mathcal{M}l$ system execute the following Unix commands (after login):

```
cd /home/mael/bin
./ml
```

These commands should startup the $\mathcal{M}ini\mathcal{M}l$ interactive system and the following lines will appear:

```
mml 0.1 The Technical University of Denmark
Based on Caml Light 0.6 and the ML Kit 1.0

mml>
```

It should now be possible to enter expressions and declarations of the $\mathcal{M}ini\mathcal{M}l$ language (see chapter 7).

Source code

The tables below lists all files that have been altered or constructed during construction of the $\mathcal{M}ini\mathcal{M}l$ system. There is a table for each subdirectory of the */src*-directory. Only the file

Makefile (the main makefile) in the */src*-directory has been altered. There is an *archive* file (a *tar* file) named *mml.tar* containing these and other files concerning the $\mathcal{M}ini\mathcal{M}l$ system in directory */home/mael/project*.

.../src/compiler/

File name	**Description**
Makefile	Makefile for the compiler
builtins.ml	Access to built-in constructors etc.
compiler.ml	The compiler
config.mlp	Configuration
errors.ml	Handling of errors
front.ml	The front-end of the compiler
lexer.mli	Interface for the lexer
lexer.mlp	Lexical analysis
location.mlp	Handling of location information
main.ml	Main source file for the compiler
match.ml	The match compiler
misc.ml	Auxiliary functions and values
par_aux.ml	Auxiliary parser functions
parser.mly	Interface for the parser
pr_decl.ml	Declaration printing
syntax.ml	Abstract syntax
tr_env.ml	Handling of the translation environment
ty_decl.ml	Typing of a declaration
ty_error.ml	Type errors
typing.ml	Type checking
version.ml	Version description
opcodes.ml	Zinc instructions
infixbas.ml	(new) Infix resolution
infixcom.ml	(new) Infix resolution
infixexp.ml	(new) Infix resolution
infixexp.mli	(new) Infix resolution
infixing.ml	(new) Infix resolution
infixpat.ml	(new) Infix resolution
infixpat.mli	(new) Infix resolution
infixtab.ml	(new) Infix resolution

.../src/lib/

File name	**Description**
Makefile	Makefile for the library
eq.mli	Interface to equality primitive
ref.mli	Interface to reference constructor
initbas.ml	(new) Initial basis, primitives
initbas.mli	(new) Interface to the initial basis
mlint.ml	(new) Functions on integers

.../src/toplevel/

File name	**Description**
Makefile	Makefile for the interactive system
do_phr.ml	Execution of a compiled phrase
main.ml	Main file for the interactive system
pr_value.ml	Value printing
toplevel.ml	The toplevel loop
toplevel.mli	Interface for the toplevel loop
version.ml	Version description

.../src/runtime/

File name	**Description**
Makefile	Makefile for the abstract machine
mlvalues.h	Representation of values
equal.c	The equality primitive
io.c	Input/output
main.c	The main source file of the abstract machine
prims.c	Primitives
sys.c	System interaction

Appendix B

Executable and Source Code for the ML Kit system

This appendix includes a description of how to test the new back-end of the ML Kit system and it also includes a list of all source code files that have been constructed or altered.

Execution file

To start a version of the ML Kit system which is integrated with the Zinc abstract machine, you must be able to access the directory */home/mael/bin* located on *idfs4* at the Technical University of Denmark. To start this version of the ML Kit system execute the following Unix commands (after login):

```
cd /home/mael/mlkit/KitZam
./StartKitAgain
```

These commands should startup the Standard ML of New Jersey compiler with the new version of the ML Kit system loaded. The following lines will appear:

```
ML-Kit 1.x. --- 28.08.94
This version generates lambda code and compiles it into zinc code
and executes it on the Zinc abstract machine.
Use 'interact ()' to toggle debugging flags.
val it = () : unit
-
```

To start the compiler type *eval ()* at the prompt. This function starts the abstract machine, compiles and executes a prelude and returns the control to the user. It is now possible to enter phrases of core Standard ML. The session is terminated with control-C. Another function, *interact ()*, is available at the Standard ML of New Jersey prompt. This function allows for toggling of the debugging flags.

Source code

The tables below lists all files that have been altered or constructed during construction of the new back-end of the ML Kit system. There is a table for each subdirectory of the *.../mlkit/KitZam*-directory and a table for the *.../mlkit/KitZam*-directory itself. There is an *archive* file (a *tar* file) named *kitzam.tar* containing these and other files concerning the new back-end of the ML Kit system in the directory */home/mael/project*.

.../mlkit/KitZam

File name	**Description**
Compiler/	Updated and new files for the compiler
FLAGS.sml	Debugging flags (signature)
Flags.sml	Debugging flags (functor)
HOOKS.sml	Hooks for the parser and the lexer
KIT_BUGS.txt	(new) Current bugs in the system
KitCompiler.sml	Linking of the compiler
ML_CONSULT_COMP.Ece.To	Files in the make project
Prelude.sml	The current prelude
Prelude.sml.gem	Another prelude
Prelude.sml.orig	The original prelude
README	(new) General information
Runtime/	The abstract machine
use_me_comp.ece.to	Initial *use* file for the make system

The ML Kit system is built by importing (*use*) the file *use_me_comp.ece.to* into a Standard ML system with the Edinburgh Library [Ber91] loaded. This file then imports (in the correct order), by use of the make system of the Edinburgh Library, all files listed in the file *ML_CONSULT_COMP.Ece.To*.

.../mlkit/KitZam/Runtime

Only files of the Zinc abstract machine of Caml Light that have been changed are listed in the table below.

File name	**Description**
Makefile	Makefile for the runtime system
config.h	Configuration
mlvalues.h	Representation of values
equal.c	The equality primitive

interp.c	The Zinc code interpreter
main.c	The main source file for the abstract machine
prims.c	List of C primitives
sys.c	System interaction
sml_prim.c	(new) Primitives for value printing
LOGFILE.txt	(new) A logfile

.../mlkit/KitZam/Compiler

File name	**Description**
COMPILER_DYNAMIC_BASIS.sml	Dynamic basis for the compiler (signature)
CompileAndRun.sml	Linking of the compilation steps
CompilerDynamicBasis.sml	Dynamic basis for the compiler (functor)
DYNAMIC_ENV.sml	Dynamic environment (signature)
DynamicEnv.sml	Dynamic environment (functor)
Evaluation.sml	A linking functor
GVARS.sml	(new) Global variables (signature)
Gvars.sml	(new) Global variables (functor)
TAG_ENV.sml	(new) Tag environment (signature)
TagEnv.sml	(new) Tag environment (functor)
ValPrint.sml	(new) Value printing (functor)
ZamBackEnd/	(new) The new back-end of the compiler

.../mlkit/KitZam/Compiler/ZamBackEnd

The files listed in the table below are all new.

File name	**Description**
BUFF_CODE.sml	Byte code buffer (signature)
BuffCode.sml	Byte code buffer (functor)
COMPILE_LAMBDA.sml	Generation of Zam code (signature)
CONFIG_ZAM.sml	Configuration file (signature)
CompileLambda.sml	Generation of Zam code (functor)
ConfigZam.sml	Configuration file (functor)
EMIT_ZAM.sml	Generation of Zinc code (signature)
EmitZam.sml	Generation of Zinc code (functor)
INSTRUCT_ZAM.sml	Zam instructions (signature)

InstructZam.sml	Zam instructions (functor)
LABELS.sml	Label generation (signature)
LAMBDA_EXP_DEBRUIJN.sml	de Bruijn lambda language (signature)
Labels.sml	Label generation (functor)
LambdaExpDeBruijn.sml	de Bruijn lambda language (functor)
OPCODES.sml	Zinc instructions (signature)
Opcodes.sml	Zinc instructions (functor)
RUN_ZINC.sml	Execution of Zinc code (signature)
RunZinc.sml	Execution of Zinc code (functor)
TOOLS_ZAM.sml	Auxiliary functions for Zam code generation (signature)
TRANSLATE_KIT_LAMBDA.sml	Translation of the typed lambda language (signature)
ToolsZam.sml	Auxiliary functions for Zam code generation (functor)
TranslateKitLambda.sml	Translation of the typed lambda language (functor)

Appendix C

Zinc instructions

The table below lists the instructions for the Zinc abstract machine [Ler90b]. *Arguments* to each of the instructions can either be *bytes*, *shorts* (two bytes) or *words* (four bytes).

Instruction	**Instruction**
CONSTBYTE (byte)	CONSTSHORT (short)
GETGLOBAL (short)	SETGLOBAL (short)
CUR (short)	SWITCH (byte, short, . . ., short)
BRANCH (short)	BRANCHIF (short)
BRANCHIFNOT (short)	POPBRANCHIFNOT (short)
BRANCHIFNEQTAG (short)	BRANCHIFEQ (short)
BRANCHIFNEQ (short)	BRANCHIFLT (short)
BRANCHIFGT (short)	BRANCHIFLE (short)
BRANCHIFGE (short)	BRANCHINTERVAL (short, short)
C_CALL1 (short)	C_CALL2 (short)
C_CALL3 (short)	C_CALL4 (short)
C_CALL5 (short)	C_CALLN (byte, short)
MAKEBLOCK (word)	MAKEBLOCK1 (byte)
MAKEBLOCK2 (byte)	MAKEBLOCK3 (byte)
MAKEBLOCK4 (byte)	TAGOF
ACCESS (byte)	ACC0
ACC1	ACC2
ACC3	ACC4
ACC5	ATOM (byte)
ATOM0	ATOM1
ATOM2	ATOM3
ATOM4	ATOM5
ATOM6	ATOM7

ATOM8	ATOM9
GETFIELD (byte)	GETFIELD0
GETFIELD1	GETFIELD2
GETFIELD3	SETFIELD (byte)
SETFIELD0	SETFIELD1
SETFIELD2	SETFIELD3
STOP	CHECK_SIGNALS
APPLY	RETURN
APPTERM	GRAB
LET	LETREC1
DUMMY (byte)	UPDATE
ENDLET (byte)	ENDLET1
PUSHTRAP (short)	RAISE
POPTRAP	PUSH
POP	PUSHMARK
PUSH_GETGLOBAL_APPLY (short)	BOOLNOT
PUSH_GETGLOBAL_APPTERM (short)	NEGINT
SUCCINT	PREDINT
ADDINT	SUBINT
MULINT	DIVINT
MODINT	ANDINT
ORINT	XORINT
SHIFTLEFTINT	SHIFTRIGHTINTSIGNED
SHIFTRIGHTINTUNSIGNED	EQ
NEQ	LTINT
GTINT	LEINT
GEINT	INCR
DECR	FLOATOP
INTOFFLOAT	EQFLOAT
NEQFLOAT	LTFLOAT
GTFLOAT	LEFLOAT
GEFLOAT	STRINGLENGTH
GETSTRINGCHAR	SETSTRINGCHAR
EQSTRING	NEQSTRING
LTSTRING	GTSTRING
LESTRING	GESTRING
MAKEVECTOR	VECTLENGTH
GETVECTITEM	SETVECTITEM
FLOATOFINT	NEGFLOAT
ADDFLOAT	SUBFLOAT
MULFLOAT	DIVFLOAT

For the *SWITCH* instruction the first argument tells the number of arguments to follow. The *FLOATOP* instruction must be followed by one of the floating point (real) instructions (*INTOFFLOAT, ADDFLOAT, SUBFLOAT* or *MULFLOAT*) and these instructions must be preceded by the *FLOATOP* instruction.

Bibliography

[AM87] Andrew W. Appel and David B. MacQueen. Standard ml of new jersey (description v. 1). Technical report, Princeton University and AT&T Bell Laboratories, 1987.

[AM91] Andrew W. Appel and David B. MacQueen. Standard ml of new jersey (description v. 2). Technical report, Princeton University and AT&T Bell Laboratories, 1991.

[App89] Andrew W. Appel. A runtime system (draft). Technical report, Princeton University, February 1989.

[App92] Andrew W. Appel. *Compiling With Continuations.* Cambridge University Press, 1992.

[App94] Andrew W. Appel. Space-efficient closure representations. In *ACM Conference on Lisp and Functional Programming*, June 1994.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Ber91] Dave Berry. The edinburgh sml library. Documentation and description of the Edinburgh SML Library, April 1991.

[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. The ml kit version 1. An implementation of Standard ML written in Standard ML, March 1993.

[Car86] Luca Cardelli. Basic polymorphic typechecking. Technical report, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1986.

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. Technical report, Edinburgh University, 1982.

[Han91] John Hannan. Making abstract machines less abstract. In *Lecture Notes in Computer Science, Functional Programming Languages and Computer Architecture, 5th ACM Conference, Cambridge, MA, USA.*, pages 618–635, August 1991.

[JL92] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. Technical report, Department of Computing Science, University of Glasgow, 1992.

[Joh93] Olof Johansson. An implementation of a type checker for μml. Technical report, University of Umeå. Institute of Information Processing. Department of Computer Science. Sweden., April 1993.

[Jon87] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, 1987.

[Kah93] Stefan Kahrs. Mistakes and ambiguities in the definition of standard ml. Technical report, University of Edinburgh, Laboratory for Foundations of Computer Science, April 1993. There is an Addenda for this paper, written June 94.

[Lab93a] AT&T Bell Laboratories. Standard ml of new jersey – base environment. Technical report, AT&T Bell Laboratories, February 1993.

[Lab93b] AT&T Bell Laboratories. Standard ml user's guide. General information about the Standard ML of New Jersey (SML/NJ) system., February 1993.

[Ler90a] Xavier Leroy. Efficient data representation in polymorphic languages. Technical report, INRIA, 1990.

[Ler90b] Xavier Leroy. The zinc experiment: An economical implementation of the ml language. Technical report, INRIA, February 1990.

[Ler92] Xavier Leroy. *Polymorphic Typing of an Algorithmic Language.* PhD thesis, INRIA, October 1992.

[Ler93] Xavier Leroy. The caml light system, release 0.6. documentation and user's manual. Available by anonymous FTP: ftp.inria.fr., September 1993.

[Mac88] David B. MacQueen. An implementation of standard ml modules. Technical report, AT&T Bell Laboratories, Murray Hill, NJ 07974, March 1988.

[Mil78] Robin Milner. A theory of type polymorphism in programming. JCSS 17, 3, pages 348–375, 1978.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, 1990.

[Pau91] Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge, University Press, 1991.

[Ŕ92] Didier Rémy. Rapport de recherche 1766. Technical report, INRIA, October 1992.

[Rob65] J.A. Robinson. A machine-oriented logic based on the resolution principle. In *JACM 12,1*, pages 23–41, 1965.

[Ses91] Peter Sestoft. *Analysis and Efficient Implementation of Functional Programs.* PhD thesis, University of Copenhagen, Department of Computer Science, October 1991.

[Tof88] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Department of Computer Science, May 1988.

[Tof94] Mads Tofte. Implementation of the typed call-by-value λ-calculus using a stack of regions. Technical report, University of Copenhagen, January 1994.

[Wad92] Philip Wadler. The essence of functional programming. Technical report, University of Glasgow, Department of Computer Science, January 1992. Presented as an invited talk at *19'th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 92.