Syntax Processing and Code Generation for a Generic Programming Language

Werner Danielczyk-Landerl
Matr.Nr.: 84 55 237

Diplomarbeit

angefertigt am
Institut für Symbolisches Rechnen (RISC-Linz)
der Johannes Kepler Universität Linz

eingereicht bei a.Univ.Prof. Dr. Heinrich Rolletschek am
14. September 1998

Themenvorschlag für die Diplomarbeit

Syntax Processing and Code Generation for a Generic Programming Language

Entwicklung und Beschreibung der Techniken zur Implementierung von Teilen des Compilers für eine funktorbasierte Programmiersprache. Die Teile sind:

- der *Parser*, der neben der syntaktischen Analyse die Aufgabe hat, Programmtexte in Syntaxbäume zu übersetzen, sowie
- der *Code-Generator*, der vom Typ-Checker annotierte Syntaxbäume für vollständig instanzierte Funktoren in linkbare Ojektdateien übersetzt.

Besonderes Augenmerk liegt auf den Techniken zur Realisierung der folgenden erwünschten Eigenschaften der zu übersetzenden Sprache:

- Die Sprache verwendet eine C-ähnliche *Kernsprache* zur Spezifizierung und Implementierung der von einem Funktor exportierten Prozeduren, Funktionen, etc. Diese Kernsprache soll jedoch *einfach austauschbar* sein. Dies erfordert eine klare Separierung der Teile des Compilers, die die Kernsprache realisiern, von denen, die das Funktor-Konzept realisieren. Insbesondere für die Verwendung eines Parser-Generators wie YACC für die Erzeugung des mit einem Kernsprachen-Parser parametrisierten Funktorsprachen-Parsers sind Strategien zu entwickeln.
- Die Sprache soll die Definition von Operatoren, sowie auch von Literalen, durch den Benutzer erlauben. Die Anwendung von Operatoren kann, wie z.B. in C++, als Anwedung spezieller Funktionen aufgefasst werden. In ähnlicher Weise können Literale als (verschachtelte) Funktionsanwendungen auf Ziffern oder Zeichen interpreiert werden. Die betreffenden Funktionen können dann von Benutzern nach Bedarf definiert werden.
- Das von der Sprache zur Verfügung gestellte Konzept des *Funktors* (parametrisierten Moduls) ermöglicht die Entwicklung generischer Bibliotheksmodule. Die Sprache stellt jedoch keine Mittel zur Erzeugung gelinkter, ausfürbarer Programme zur Verfügung. Vollständig instanzierte Module sollen von Standard-Sprachen wie C, C++ oder FORTRAN aus verwendbar sein. Dies erfordert die Erzeugung von mit Standard-Linkern bearbeitbaren Objektdateien durch den Code-Generator.
- Für ausführbare Spezifikationen von Funktionen sollen Default-Implementierungen generiert werden. Aus Ein-/Ausgabespezifikationen von Prozeduren oder Funktionen soll Code für Laufzeitüberprüfungen erzeugt werden.

Kurzfassung

Wir beschreiben die entwickelten und verwendeten Techniken zur Implementierung des Parsers und Code-Generators für eine Programmiersprache, die Funktoren, d.h. parametrisierte Module, zur Entwicklung generischer Bibliotheken nutzt. Eine imperative Kernspache dient zur Implementierung der von einem Funktor bereitgestellten Elemente.

Der Parser erzeugt einen Syntaxbaum, der den geparsten Quelltext repräsentiert. Mehrere Unterparser werden zum Parsen von bestimmten syntaktischen Domains aufgerufen (und rufen sich gegenseitig auf). Insbesondere sind die Parser für die syntaktischen Domains der C-ähnlichen Kernsprache von den Parser-Teilen für die eigentliche Funktorsprache separiert. Dies ermöglicht bei Bedarf einen einfachen Umstieg auf eine andere Kernsprache. Der Parser versteckt die in Ausdrücken der Sprache erlaubten Operatoren und Literale als rein syntaktische Spracheigenschaften vor anderen Compiler-Teilen, macht sie aber andererseits benutzerdefinierbar.

Der Code-Generator erzeugt C++ (im wesentlichen C) Code für vollständig instanzierte Funktoren. Dieser wird vom GNU C++ Compiler in Objectcode übersetzt, der zu C oder C++ Programmen gelinkt werden kann. Die Funktorsprache erlaubt die Definition von Konstruktoren und Destruktoren für benutzerdefinierte Typen. Der für Ausdrücke erzeugte Code sorgt für die korrekte Lebenszeit und entsprechende Destruktion aller temporären Werte. Aus Funktionsspezifikationen werden Default-Implementierungen für Funktionen erzeugt. Abhängig von Art und Typ von Parametern werden passende Übergabemechanismen ausgewählt. Dies ist möglich, da die Semantik der von der Funktorsprache zur Verfügung gestellten Parameterarten nicht durch bestimmte Übergabemechanismen festgelegt ist. Schließlich enthält der erzeugte C++ Code Debugging-Information, die das Verfolgen des Ablaufes von Quelltexten der Funktorsprache mit herkömmlichen Quelltext-Debuggern ermöglicht.

Abstract

We describe the techniques developed and used for implementing the parser and code generator for a programming language that exploits functors, i.e., parameterized modules, for developing generic libraries. An imperative core language is used for describing the implementation of the entities provided by functors.

The parser generates a syntax tree representing the parsed source text. Several sub-parsers are called (and call each other) for parsing certain syntax domains of the language. In particular, the parsers for the syntax domains of the C-like core language are separated from the actual functor language parts of the parser. This allows to easily switch to another core language if desired. The parser hides the operators and literals allowed in the expressions of the language as purely syntactic features from the other parts of the compiler, and makes them, on the other hand, user-definable.

The code generator produces C++ (essentially C) code for fully instantiated functors. This is compiled by the GNU C++ compiler to object code that can be linked to C or C++ programs. The functor language allows the definition of constructors and destructors for user-defined types. The code generated for expressions serves for the correct life-time and appropriate destruction of all temporary values. Default implementations for functions are generated from function specifications. Appropriate parameter passing mechanisms are chosen depending on parameter modes and types. This is possible since the semantics of the parameter modes provided by the functor language are not specified in terms of certain passing mechanisms. Finally, debugging information is included in the generated C++ code that allows tracing the execution of functor language source texts with usual source level debugging tools.

Für Brigitte

Contents

APPENDICES

Chapter 1

Introduction

We report on work done in the frame of the HPGP (*High Performance Generic Programming*) project [HS96]. This project pursues the development of a programming environment for the construction of generic mathematical libraries with high-performance requirements. The author was responsible for the development of the syntax processing part of the compiler for the underlying functor-based programming language, as well as for the code generation part.

1.1 The HPGP Project

The HPGP project employs *functors* (parameterized modules) as the basis for developing software libraries that provide solutions in various mathematical domains. Module parameterization as the basic device for building large software systems was introduced by the functional language SML [MTH90, Ber93] and by various special purpose computer algebra systems [Scr88, Jen92].

Work in this direction was inspired by previous attempts to build generic mathematical libraries [HN+94, WB93] that were seriously hampered by a lack of appropriate language support. The main-stream in software engineering currently concentrates on the object-oriented notion of inheritance for building reusable software [GHJV95]. However, it is difficult to model mathematically axiomatized structures by inheritance; parameterized structures offer a much more natural framework (cf. the project proposal [HS96]).

1.1.1 The Programming Environment

In industrial programming languages, parameterized program units have only found limited support, see, e.g., Ada [ANS83, ISO94] or Modula-3 [Nel91]. A major exception are *templates* in C++ [Str97] as the basis of the Standard Template Library [MS96]. However, C++ templates do not provide means for specifying information about their parameters. Stroustrup argued in [Str88] that this would reduce the flexibility of the parameterization facility without increasing its safety. Thus the body of a template can not be type-checked on its own. Later

Stroustrup actually perceived the importance of constraints for type parameters in readability and early error detection and proposed to use the required operations in a dummy function, or to use inheritance from a base class instead [Str94]. Another problem is that many C++ implementations are not able to share code for template instances among different users of a template library. Both problems make templates difficult to use in large scale mathematical software development [CH+94].

SML's functors, on the other hand, are statically type-checked. However, they are compiled to function tables that are looked up at runtime with corresponding performance losses. Furthermore, the functor concept is bound to a relatively high level, mostly functional core language. And finally, SML does not provide means for specifying the *behavior* of components apart from their syntactic interfaces and type signatures.

The goal of the HPGP project is to provide an environment for the multi-user development of generic mathematical libraries without compromising the performance of the generated code based on the following concepts:

- A *low-level core language* is utilized for writing the executable entities (procedures, functions) embedded in and exported by modules. This core language provides no powerful built-in concepts that would require a complex run-time system. All abstractions necessary for convenient programming should already be built in the programming environment itself.

  Currently a language called Cora (essentially a simple subset of C) is used as core language. However, the HPGP implementation of the functor concept should not be bound to a particular core language. Thus it should be easy to exchange Cora by any other suitable language.

- The language Mona on which the environment is based provides *powerful abstraction facilities* for constructing generic software libraries. *Specifications* define the syntactic and semantic interface of modules. *Functors* describe how new modules can be constructed from other modules. *Modules* result then from applying functors.

  Currently Mona is restricted to first-order functors, i.e., functors cannot be components of modules or passed as arguments to other functors. It is intended to further develop the system into a higher-order framework [CM94, Ler95, Bis95] after the current status is consolidated.

- The implementation is carefully designed to make sure that *abstraction is not punished by runtime overhead*. Information about the implementation of an entity is transparently propagated by the type-checking and instantiation mechanisms of the compiler through all levels of module abstractions. Thus ultimately the same code is generated as without using the abstraction.

  To avoid code explosion, no duplicate code is produced for equivalent functor instantiations.

- *Support for project management* is provided by an elaborate package concept, as well as by dependence control mechanisms built into the compiler. *Packages* group specifications, functors, modules and other packages together. They may be jointly developed and used by multiple persons in a shared environment. The system is designed such that code is never unnecessarily duplicated, permissions are set correctly, and dependencies between different units are maintained automatically.

It is planned to extend the environment by a revision control system for developed libraries.

1.1.2 The Compilation Process

A Mona source file is first translated to a *syntax tree* by the *parser*. This syntax tree is what is actually kept as *source*: it is stored in binary form in a central repository. A *pretty printer* is provided for printing syntax trees in program text form. Thus it is not necessary for the user to keep source files as human-readable versions of the sources stored in the repository.

In the next stage the syntax tree is processed by the *type checker*. The type checker compiles all information known about every entity exported by the translation unit into a *symbol table*. This is stored in the repository as the *code* produced for the unit. For specifications, the information stored with a symbol includes all *axioms*, i.e., properties, attached to the corresponding entity. For functors, it includes complete implementation information.

For functors, the type checker additionally replaces all identifiers in the processed syntax tree by references to the corresponding symbol table entries and adds type information at certain points in the tree. This *annotated syntax tree* is stored together with the symbol table.

For modules, object code and a C header file with prototypes for the entities exported by the module are additionally stored in the repository. Modules are described as applications of functors to other modules. The object code and header file for a module is produced by a call of the *code generator* with an instantiated version of the annotated syntax tree of the applied functor.

The object code for a module contains definitions only for entities implemented in the instantiated functor, not for entities defined in parameter modules. It has to be linked with the object files of the parameter modules, which in turn need the object files of their parameters, and so on. An *exporter* is provided, which generates the list of the pathnames of all object files in the repository required for using a certain module, as well as header file that contains an `#include` directive with the pathname of the module header file in the repository.

The type checker is in fact called via the *dependence control facility*. This facility checks whether the code (symbol table) for all units required by the unit to be compiled is up to date (w.r.t. the units in turn required) and triggers re-compilation of the corresponding sources (syntax trees) if necessary before actually invoking the type-checker. Thus dependencies are resolved recursively without the necessity of any user-intervention or assistance.

On the other hand, before code (symbol table and object code) for a module is generated, it is checked whether up-to-date code for an instance of the applied functor with the same module arguments is already available in the repository. Thus no module code is duplicated.

1.1.3 Implementation

The Mona compiler is implemented in a subset of C++ [Lan96], compiled with g++, the GNU C++ compiler. Compliance with the imposed restrictions of C++ usage is checked by a separate parser. The used subset simulates *modules* by C++ classes with only `static` members. No object-oriented features are used. This should ease a possible later translation of the compiler sources to Mona.

The parser is implemented using FLEX and BISON, the versions of LEX and YACC provided by the GNU project. It consists of a set of cooperating sub-parsers as described in the following Syntax Processing part of this report.

The complete sources, as well as other information related to the project, can be seen at the project Web page, whose URL is

`http://www.risc.uni-linz.ac.at/projects/basic/hpgp`.

1.2 Outline

The results of our work in the frame of the HPGP project are presented in two parts corresponding to the parts of the Mona compiler we were concerned with. The source code of the developed compiler components is not included in this report but can be seen at the Web page mentioned above.

1.2.1 Syntax Processing

Chapter 2, *Syntax Processing Overview*, gives an overview of the issues involved in processing the concrete syntax of Mona. A short description of the grammar of Mona is provided, the strategies used to develop the structure of the syntax trees generated by the parser are discussed, and the implementation of parser and pretty printer is sketched. Also the special treatment of comments in Mona is described here: Comments in Mona sources are not filtered out by the scanner, but are part of the parser syntax. The parser has to include them in the generated syntax tree, so that the pretty printer is able to reproduce the comment text and its position.

Separate chapters are dedicated to the more specific syntax processing requirements of the HPGP project and the techniques developed to fulfill them. These requirements are:

- an easily exchangeable core language and hence a parser with easily exchangeable core language part, and
- a user-definable interpretation of operators and literals to make these syntactic features usable for user-defined types.

Chapter 3, *Parameterized Parsers*, describes the techniques developed to make the Mona parser independent of a particular core language. The use of parameterized grammars is demonstrated, as well as our strategy for using BISON to develop parameterized parsers. Furthermore, the actual structure of the Mona parser that resulted from applying these techniques is presented.

Chapter 4, *Special Parser Features*, finally, describes how the parser makes operators and literals user-definable, while hiding the existence of these syntactic features from other compiler components.

Some details are provided in appendices:

- Appendix A contains the complete grammar of Mona,
- Appendix B describes the lexical syntax of Mona, and
- Appendix C specifies the structure of the syntax trees by means of a grammar for the *abstract syntax* of Mona.

1.2.2 Code Generation

Also the code generation part starts with an overview chapter: Chapter 5, *Code Generation Overview*, describes the annotated syntax trees that are the input for the code generator, as well as the coarse structure of the generated code, including the technique used to handle inlining of function and procedure definitions. Furthermore, the different phases of code generation are discussed: it was necessary to split the code generator functionality into these phases because of the possibility of cyclic module dependencies, which require code for several modules to be produced at once. Finally, the functionality and implementation of the exporter for Mona modules is sketched in this chapter.

Chapter 6, *The Code in Detail*, gives a detailed description of the C++ code generated by the Mona code generator for the different constructs of Mona. This includes how expression evaluation is resolved into C++ code using temporary variables for intermediate results of expressions, so that all temporary values have the required life-time and are destroyed afterwards by a call of the appropriate destructor.

Chapter 7, *Special Code Generator Features*, finally, describes how information provided in module specifications is exploited by the code generator for producing default implementations or run-time checks of input/output conditions. Furthermore, several optimizations are described that lead to better code than described in Chapter 6 being actually generated in certain cases. Most importantly, since the semantics of parameter modes are not specified in terms of concrete parameter passing mechanisms, the code generator is able to choose appropriate mechanisms depending on the parameter type. As a last point, it is described how debugging information is added to the C++ code that allows to trace the execution of Mona sources with source level debugging tools.

Again, additional information is provided in appendices:

- Appendix D lists all built-in types and operations available in Mona together with the code produced for them, and
- Appendix E contains the complete sources of a simple Mona module, together with the C++ source of the code generated for the module and a demonstration of its usage.

Acknowledgements

The author wishes to express his thanks to Wolfgang Schreiner, the director of the HPGP project, for the opportunity to do interesting work in a pleasant (and thus efficient and successful) project framework, as well as for much helpful input during the preparation of this thesis.

Thanks go also to the other coworkers in the project, Mircea Marin and Wolfgang Stöcher, for a pleasant and fruitful collaboration.

The HPGP project was supported by the Austrian Science Foundation FWF grant P11414-ÖTE "HPGP—High-Performance Generic Programming".

Part I

SYNTAX PROCESSING

Chapter 2

Syntax Processing Overview

Only two components of the HPGP system are concerned with the concrete syntax of the Mona language: the *parser* and the *pretty printer*.

The parser uses the *grammar* of Mona to analyze the syntactic structure of a Mona source file. If the source complies with the grammar, a *syntax tree* representing the source is generated. Otherwise, error messages are produced, and the failure is indicated to the caller.

The syntax tree generated by the parser is then used as input for the following type checking pass of the compiler. A binary representation of the tree is also stored in the central repository of the system, for use in later recompilations triggered by the dependency control mechanisms of the compiler.

To avoid the necessity to store also the human-readable source in the repository, a pretty printer is provided. It gets as input a syntax tree and produces a corresponding textual representation as a source file complying with the Mona grammar. The pretty printer provides thus the inverse mapping of what the parser does: pretty printing a syntax tree and parsing the resulting source file yields the original syntax tree.

In the following sections, we will introduce the grammar of Mona as well as the structure of the syntax tree produced by the parser. Also the implementation of the parser and pretty printer is sketched. This includes a description of our way to deal with *comments* in Mona. Comments are not considered as an additional type of white space in Mona sources, but occur in the grammar. The parser includes them in the generated syntax tree, so that the pretty printer is able to reproduce the comment text and its position. However, this should not slow down processing of the tree by compiler components not interested in the comments.

The developed techniques for implementing the Mona parser in a way that allows to easily exchange its core language part, as well as for making operators and literals independent of certain built-in types, are described in detail in separate chapters.

2.1 Grammar

The complete grammar of Mona is given in Appendix A, clearly separated into a functor language and a core language part. The strategies used to develop a functor language grammar independent of a certain core language are demonstrated with a simple example in Section 3.1.

We do not intend to discuss all language features here, but describe only the coarse structure of the language. More information on constructs of the language is contained in the Code Generation part of the report, but for a detailed treatment see the HPGP manual [SDLMS98]. For the semantic issues involved in designing a functor language, see also [Stö97].

2.1.1 Translation Units

Mona knows three different kinds of translation units: *specifications*, *functors* and *modules*. The term 'functor' is used for a module parameterized with other modules. A Mona functor unit describes correspondingly the implementation of a module dependent on certain parameter modules.

A module results then from *instantiating* the functor, i.e., from supplying argument modules for the functor. This is done in Mona module units, which describe the construction of modules from other modules by applying functors.

The basis of such a construction is formed by parameterless functors, which do not require other modules as arguments. Still, Mona distinguishes them from modules: they have to be explicitly instantiated in a module unit by supplying an empty module argument list.

Specifications, finally, describe the *interface* of modules, i.e., which items a module exports and what properties these items have. Mona provides a rich specification language, allowing to use first order predicate logic formulae to specify input and output conditions for procedures and functions, values of constants, etc.

Properties in specifications and procedure or function implementations in functors are written in the form of expressions and commands of an exchangeable imperative core language. For the HPGP project, a C-like core language named Cora was used.

Functors allow a few other ways to define items than just to give an implementation in the core language:

- A `new` definition introduces a new type with the given representation. The defined type is considered to be different from all other types, including its representation type.
- An `alias` definition gives just a new name to an item defined elsewhere.
- An `extern` definition introduces a name for an item defined externally, not in Mona. This produces external references in object files for instances of the functor, which have to be resolved at link time.

- A `builtin` definition introduces a name for a built-in item. The built-in types and operations provided by Mona are not visible in functors without explicit `builtin` definitions. A list of all built-in items available for use in `builtin` definitions is provided in Appendix D.
- A `default` definition of a function means that the specification of the function should be translated into a default implementation.
- A `forward` definition, finally, introduces just the name and interface of the item. It is used for writing recursive definitions.

Definitions with keyword `builtin` usually should not occur in user code. All built-in types and operations are provided by modules in a `Standard` package. Only these modules are built by means of `builtin` definitions.

Certain core language constructs require values of some built-in type. The `if` or `while` commands, e.g., as well as conditional expressions, expect a condition of the built-in boolean type. To allow also user-defined types in such places, conversion functions are applied. These have to be specified with special type axioms:

- `axiom type` *T* `truth` *f* specifies that the function *f* maps a value of type *T* to the built-in boolean type,
- `axiom type` *T* `case` *f* specifies that the function *f* maps a value of type *T* to the built-in integer type (which is required in `switch` commands), and
- `axiom type` *T* `address` *f* specifies that the function *f* maps a value of type *T* to the built-in address type (required in dereferencing expressions).

For type-checking these axioms, the information that the result type of the function *f* is the expected built-in type is required from the *specification* of this type. Therefore, a type can not only be *implemented* but even *specified* as `builtin` (see the grammar rule for *typeSpec* in Appendix A). Again, this should only occur in the specifications of the `Standard` modules.

2.1.2 Commands

The commands of Cora are C statements, with a few exceptions:

- In command blocks, definitions of local variables or constants and commands may be freely intermixed as in C++. However, the syntax of the local definitions is fixed by the functor language.
- The syntactic form of the `switch` command is restricted: only a single command is allowed after each sequence of `case` labels. A command sequence in this place has to be explicitly enclosed in braces.
- There is no `for` command.
- Labels and the `goto` statement are not supported.

- There is no empty command. An empty block must be used instead.

The command domain is additionally extended with syntax for explicit constructor and destructor calls by the functor language.

2.1.3 Types and Expressions

The compound types that can be constructed in Cora are records and unions. Their syntax differs from the C syntax for syntax error recovery reasons. No pointer or array type constructors are required from the core language, since these can already be implemented in the functor language.

The expressions allowed in Cora are the same as in C, with again a few exceptions:

- No type casts are allowed.
- There is no comma-expression.
- There is an additional infix operator '=>' for logical implication.
- There is an additional dereferencing operator '`@`'.
- There is an additional '`alignof`' operator.

In specifications, the expression domain is extended by the functor language with quantifiers, `let` blocks, and an `old` operator for referring to the input values of parameters in the output conditions of procedures. In functor units, on the other hand, the functor language allows a `repr` operator for casting an expression of a `new` type to its representation.

2.1.4 Identifiers

Two different terminal classes for identifiers are used in the Mona grammar. Their lexical syntax is described in Appendix B. The first one, *`unitId`*, is provided by the functor language. The second one, *`coreId`*, provided by the core language, allows *operator names*, *character names* and *digit names*. These are used for making operators and literals user-definable. See Section 2.3 for more information.

2.1.5 Comments

In Mona, comments are not just an additional type of white space in the lexical syntax, but are part of the parser syntax. They are included in the syntax tree that is constructed by the parser, so that they can be reproduced by the pretty printer. The goal of devising a grammar containing comments was to get an attachment of comments to syntax constructs, so that, e.g., in case of a compilation error, the comments attached to the construct causing the error can be included in an error message. We shortly sketch our results.

Comments in Mona source files start with two slashes "//" and last to the end of the line. However, the scanner collects already consecutive comment lines to blocks. The result is communicated to the parser in form of either *separate comment* (`scmt`) or *trailing comment* (`tcmt`) tokens, depending on whether the first comment line of the block is on a line of its own or 'hangs' on some other syntax construct (cf. Appendix B).

Potentially every Mona syntax construct may have *two* comments attached to it: one coming before the construct (a *comment head*) and one coming behind it (a *comment tail*). A sequence of separate comments is accepted as comment head, whereas a single trailing comment is allowed as comment tail, except at the end of a source file, where a single separate comment is allowed as comment tail of the Mona unit contained in the file.

To make the grammar not too complicated, however, in fact only a subset of the syntax constructs of Mona is allowed to carry comments, and for some constructs of this subset only a comment tail is allowed.

Concerning the placement of comment tails, the situation is a bit more complicated. They are not really always allowed *behind* the syntax construct they get attached to, since this would, e.g., make it impossible for

```
while (expression) command // comment tail
```

to have a comment tail for both the entire `while` command, and its component, `command`. Therefore we decided for compound constructs to accept a comment tail behind the *head* of the construct (before its *components*) instead of at the very end. This resolves the above problem by allowing

```
while (expression) //comment tail for while
  command // comment tail for command
```

On the other hand, it implies, e.g., for command blocks like

```
{ // comment tail for block
  command; // comment tail for command
  ...
}
```

that there can be no trailing comment behind the closing brace. On the expression level, the comment tail for an infix expression must come for the analogous reason directly after the operator instead of at the very end, as, e.g., in

```
summand1    // comment tail for summand1
  +         // comment tail for entire sum
  summand2  // comment tail for summand2
```

There are still some ambiguities left, e.g., in

```
return f(a) // comment
```

it is not clear whether the comment tail belongs to the expression or to the return command. These are resolved by attaching the comment always to the innermost construct, i.e., in the above example, to the expression.

2.2 Syntax Trees

The structure of the syntax trees to be generated by the parser was derived from the grammar by removing all terminal symbols and considering the possible derivation trees of the resulting *abstract syntax* grammar as the possible Mona syntax trees. As an optimization, rules whose right hand side was just a list of alternatives all consisting of a single nonterminal symbol, like the one for $spec$, where eliminated by replacing every occurrence of their left hand side by the right hand side. This eliminates some nodes from the syntax tree that are not required for easy processing. The resulting abstract syntax is shown in Appendix C. The mentioned elimination is not really carried out there, since this would make the grammar quite unreadable. Instead, the affected rules are declared as 'macros' that have to be 'expanded' before interpreting the grammar.

Optional items are treated specially in deriving the syntax tree from the abstract syntax: an optional item "$[n]$" occurring in the grammar represents a tree with either an $opt1$ root label and n as subtree, or a node with label $opt0$ and no subtrees. This corresponds to replacing "$[n]$" by "$(a \mid b)$", for some new nonterminals a and b with rules

$$a := n.$$
$$b := \varepsilon.$$

However, it avoids the need for inventing new label names a and b for every optional item by reusing $opt1$ and $opt0$.

At some points, the abstract syntax as given in Appendix C deviates from the concrete syntax given in Appendix A. In some cases this is due to the fact that the abstract syntax was in fact derived from older versions of the concrete grammar. Since almost all components of the Mona compiler depend on the structure of the syntax trees, we did not do changes there just for cosmetic reasons. During development, also some easy optimizations of the tree were done in an unsystematic fashion. The `if` command of the core language command domain, e.g., was replaced by two alternatives $ifCmd$ and $ifElseCmd$, the first one without and the second one always with `else` part. This eliminated the need for the $opt1$ or $opt0$ node representing the optional `else` part.

Other manipulations were done with the goal of achieving an easy to process tree structure. Two kinds of nodes can be distinguished in the syntax trees resulting from the used abstract syntax:

- Nodes with a fixed number k of sons, represented by rules of the form

  $$l := r_1 \cdots r_k. \qquad (k \geq 0)$$

  where each r_i represents a single subtree.

- Nodes with an arbitrary number of sons, but all of the same kind, i.e., with the same list of alternatives for all sons. These are represented by rules of the form

  $$l := \{r\}.$$

where again r represents a single subtree.

This makes processing of the tree easier than with general structures resulting, e.g., from right-hand rule sides like "$\{a\}\ b$". For such a rule, one would always have to check whether any of the possible trees represented by b is at hand while traversing the son list. With a right-hand rule side of the form $\{r\}$, one only has to check for the end of the list. However, to get such a restricted tree, additional nonterminals have to be introduced, namely:

- for sequences in the middle of the right-hand side of rules, so that the sequence-enclosing braces appear only as top level operator of right-hand sides, and
- for concatenations appearing as alternatives, so that lists of alternatives are always of the form "$(r_1 \mid \cdots \mid r_k)$" with each r_i representing a single tree.

For the above described handling of optional items to work, additional nonterminals also have to be introduced for concatenations appearing optionally, since the $opt1$ node is expected to have just a single son.

The value expression domain of the abstract syntax differs quite much from its concrete counterpart. The deep nesting of the domain in the concrete grammar, together with declared operator precedences and associativities, serves the purpose of unambiguously recognizing the structure of the textual representation of expressions. This is of course not needed in syntax trees, which anyways represent a unique structure. Thus, the value expression domain of the abstract syntax is completely flat. Furthermore, some parts, like literals or most operator expressions, are missing in the abstract syntax. They are eliminated by the parser, as discussed in the following section.

The following special leaf nodes are used in syntax trees:

- Identifier nodes `id`, which refer to entries of the *identifier table* that contains the textual representations of all identifiers used in a Mona source. All kinds of identifiers (the restricted `unitIds`, as well as the extended `coreIds`) are represented by `id` nodes. The distinction between different identifier classes is only used by the parser; see the following section.
- Line number nodes `lineNr`, which contain source line numbers. They are produced by the parser to allow compiler components processing the syntax tree to refer to source lines in error messages.
- Comment nodes `lcmt` (*leading comment*) and `tcmt` (*trailing comment*). A comment head as described in the previous section is represented by a sequence of `lcmt` nodes. A comment tail is represented by a `tcmt` node. The comment nodes refer to entries of the *comment table* that stores the comment texts of Mona sources.

Comments are treated in a special way. As already mentioned, they are included in the tree. However, this should not slow down processing, i.e., it should be possible to traverse the tree without having to traverse comment sequences attached to some constructs. We achieved this by appending all comment nodes attached to a certain construct to the end of the list of sons of the node representing this construct. A compiler module not interested in comments can thus stop traversing a son list as soon as a comment node is found. Line number nodes are handled similarly: if a line number node is generated by the parser for some syntax construct, it is appended to the son list of the construct more or less as a first comment node.

2.3 Parsing

The parser was implemented using BISON, the GNU version of YACC. One of the main goals of the HPGP project was to develop a functor-based programming environment independent of a fixed core language. Developing a parser for the functor language independent of the used core language required to split the parser for the core language Cora into parts to be called from the functor language parser, like a part for parsing expressions and a part for parsing commands. The Cora parts of the parser can thus be exchanged by sub-parsers for a different core language. For such a cooperation between parsers, it might be necessary to communicate lookahead tokens. The calling parser, e.g., might need a lookahead just to decide that another parser has to be called. The called parser, however, needs the lookahead consumed by the caller as first token. Communicating tokens between BISON generated parsers is not easily possible. We solved this with a slight change of the parser template used by BISON. The corresponding techniques, as well as the actual structure of the Mona parser, are described in detail in Chapter 3.

Translating the EBNF grammar listed in Appendix A into a BISON grammar is easy: sequences were transformed into left recursive rules, and options into alternatives allowing the empty string. Some parsing conflicts due to the LALR(1) parsing mechanism used by BISON had to be resolved by easy grammar transformations.

The lexical analyzer used by the parser is a FLEX-generated scanner. The FLEX scanner specification is easily derived from the specification of the lexical syntax given in Appendix B. For the identifier and comment terminal classes, the attribute value communicated from the scanner to the parser is just the corresponding input text. For literals, a syntax tree is constructed as attribute, as discussed below.

Constructing the syntax trees in the actions to the BISON grammar rules provided no problems. Some features of the concrete syntax, like literals, many operators, as well as two different terminal classes for identifiers, are already completely resolved by the parser. They have thus no counterpart in the syntax trees generated by the parser.

The two different identifier classes are used by the parser on the one hand to guarantee that identifiers can be used as file names where this is needed, by allowing only the restricted form *unitId* in such places. The extended form *coreId* provided by the core language, on the other hand, is used to allow user defined operators and literals. The parser translates occurrences of most of the operators available in Mona into applications of corresponding *operator names*. The syntax trees for these applications are then generated. Thus, these operators do not show up in syntax trees. Operator names are one possible form of identifiers allowed by the definition of `coreId`. They can thus be used as names of user-defined functions like any other identifier. If a definition for such an operator name is given with the fitting number of parameters, this effectively defines the corresponding operator. In a similar manner, literals are translated into expressions built by applying certain operator names to *character names* or *digit names* representing the atomic components of the literal. This time, the translation is already done by the scanner. The details to all these issues are collected in Chapter 4.

As a final task, the parser has to add line number nodes to the syntax tree. Otherwise, the following passes of the compiler would not be able to refer to positions in source files in error or warning messages. For this purpose, the scanner keeps a line counter up to date.

2.4 Pretty Printing

Producing syntactically correct program text corresponding to a given syntax tree is straight forward. The structure of the pretty printer arises naturally from the structure of the syntax trees. No sophisticated formatting is done currently: line breaks are inserted at fixed points in the grammar, as well as in certain places like before parameters in parameter lists, or before components of expressions, if a certain column number was already reached before. Still, the produced output is acceptable in most cases.

For better readability, the pretty printer undoes the transformations applied by the parser to operators and literals (see Chapter 4. If an operator name is found as function name in an application tree, the tree is traversed before any output is produced, to check whether it could represent an operator expression or a literal. This avoids the rather lengthy operator names in the output (especially the expressions corresponding to literals tend to get very long).

The pretty printer provides also functions to print certain sub-trees, like trees representing long identifiers, expressions, or commands. These are used in the type checker to output error messages. This requires that the pretty printer is also able to deal with trees already updated by the type checker, which replaces type expressions and identifiers by type or symbol information. These *annotated* syntax trees are discussed in Section 5.1. Furthermore, also types, i.e., entries of the type table created by the type checker, can be printed. This includes also function and procedure types.

Finally, the pretty printer on request also updates the syntax tree by adding line number nodes according to its own formatting. This is used in the code generator for adding debugging information referring to Mona source files to object files. See Section 7.4 for details.

Chapter 3

Parameterized Parsers

The Mona functor language should extend an imperative core language by constructs to specify and build functors. Developing a parser for the functor language independent of the used core language required to split the parser for Cora, the core language we used in the project, into parts to be called from the functor language parser. This, in turn, required to split also the functor language parser and to provide sub-parsers to be called from core language parts, e.g., a parser for the extensions of the core language expression domain used in module specifications. The resulting parser structure allows to exchange the Cora parts of the parser by sub-parsers for a different core language.

Here we describe the techniques used to split parsers into cooperating parts. We start with showing how a grammar for a language extension can be made independent of the core language by using *syntax domain parameters* and describe then how *parameterized parsers* for such parameterized grammars can be implemented using BISON. Predicate logic formulae are used as an example for a language that extends (various syntactical representations of) the core language of propositional logic formulae.

Afterwards we take a brief look at what is required from a grammar such that the approach described before is applicable and discuss then what troubles might arise in writing parsers for grammars with domain parameters, using the techniques introduced before, and how they might be solved. After a few comments on the impact of the discussed techniques on the possibilities for syntax error handling with BISON, we describe finally the actual structure of the Mona parser.

3.1 Parameterized Grammars

In this section, we demonstrate the problems encountered when developing a grammar extending an unknown core language. These problems can be solved using a *parameterized grammar*, i.e., a grammar with unspecified syntax domains, which can be set later on to the domains of the chosen core language.

3.1.1 Extending a Grammar

We are going to examine the differences between grammars for first order predicate logic formulae derived from two different grammars for propositional logic formulae, namely from:

form := *infix*.

infix :=
  prefix
| *infix* "`=>`" *infix*
| *infix* "`||`" *infix*
| *infix* "`&&`" *infix*.

prefix :=
  `id`
| "`!`" *prefix*
| "`(`" *infix* "`)`".

form := *appl*.

appl :=
  `id`
| `nand` "`(`" *appl* "`,`" *appl* "`)`".

Both grammars describe formulae in a C-style syntax. The left-hand side grammar uses infix operators for implication, disjunction and conjunction, a prefix operator for negation, and parentheses to supersede operator precedences or associativity, which have to be assigned to the infix operators to make the grammar unambiguous. The right-hand side grammar allows only NAND operations in function application notation.

Extending these core language grammars to grammars for first order predicate logic formulae, which additionally allow quantifiers and use predicates instead of simple identifiers, yields:

form := *infix*.

infix :=
  quan
| *infix* "`=>`" *infix*
| *infix* "`||`" *infix*
| *infix* "`&&`" *infix*.

quan :=
  prefix
| `forall` "`(`" *ids* "`)`" *quan*
| `exists` "`(`" *ids* "`)`" *quan*.

prefix :=
  predicate
| "`!`" *quan*
| "`(`" *infix* "`)`".

form := *quan*.

quan :=
  appl
| `forall` "`(`" *ids* "`)`" *quan*
| `exists` "`(`" *ids* "`)`" *quan*.

appl :=
  predicate
| `nand` "`(`" *quan* "`,`" *quan* "`)`".

The auxiliary syntax domains *predicate* and *ids* common to both versions are defined as:

predicate := `id` "`(`" [*ids*] "`)`".

ids := `id` {"`,`" `id`}.

One can see that the main part of the extension, the syntax domain $quan$, is very similar for the two core language versions, but not quite the same. Furthermore, also the top level domain $form$ is a core language domain, namely $infix$, in one case, and an extension domain, namely $quan$, in the other. What we want to have is *one* extension grammar (the functor language grammar in case of the HPGP Project), parameterized with syntax domains for a core language.

3.1.2 Making the Parts Independent

A grammar for predicate logic formulae with parameters for the core language syntax domains could look like follows:

$$
\begin{array}{l}
form := F_{form}. \\[6pt]
quan := \\
\quad\;\; P_{quan} \\
\quad \mid \texttt{forall}\ \text{“(”}\ ids\ \text{“)”}\ quan \\
\quad \mid \texttt{exists}\ \text{“(”}\ ids\ \text{“)”}\ quan.
\end{array}
$$

This grammar depends on the domain parameters F_{form} and P_{quan}. Defining these as

$$
\begin{array}{ll}
F_{form} := infix. \qquad\qquad & F_{form} := quan. \\[6pt]
P_{quan} := prefix. & P_{quan} := appl.
\end{array}
$$

we get grammars defining the same languages as the ones from the previous subsection.

Parameterizing also the core language parts of the grammar allows to reconstruct a grammar for the core language by appropriate definitions of the domain parameters. This is required, e.g., for Mona core language expressions, which are used in functor implementations, but are extended by quantifiers and `let` blocks in specifications. Parameterized versions of the two core language grammars given in the previous subsection are:

$$
\begin{array}{ll}
form := F_{form}. & form := F_{form}. \\[6pt]
infix := & \\
\quad\;\; P_{infix} & \\
\quad \mid infix\ \text{“}\texttt{=>}\text{”}\ infix & \\
\quad \mid infix\ \text{“}\texttt{||}\text{”}\ infix & \\
\quad \mid infix\ \text{“}\texttt{\&\&}\text{”}\ infix. & \\[6pt]
prefix := & appl := \\
\quad\;\; A_{prefix} & \quad\;\; A_{appl} \\
\quad \mid \text{“}\texttt{!}\text{”}\ P_{prefix} & \quad \mid \texttt{nand}\ \text{“(”}\ F_{appl}\ \text{“,”}\ F_{appl}\ \text{“)”}. \\
\quad \mid \text{“(”}\ F_{prefix}\ \text{“)”}. &
\end{array}
$$

With the following domain parameter definitions we again get grammars for predicate logic formulae as introduced in the previous subsection:

$F_{form} := infix.$ $\qquad F_{form} := quan.$

$P_{infix} := quan.$

$P_{quan} := prefix.$ $\qquad P_{quan} := appl.$

$A_{prefix} := predicate.$ $\qquad A_{appl} := predicate.$
$P_{prefix} := quan.$ $\qquad F_{appl} := quan.$
$F_{prefix} := infix.$

The following definitions, however, yield grammars for the two core language versions:

$F_{form} := infix.$ $\qquad F_{form} := appl.$

$P_{infix} := prefix.$

$A_{prefix} :=$ `id`. $\qquad A_{appl} :=$ `id`.
$P_{prefix} := prefix.$ $\qquad F_{appl} := appl.$
$F_{prefix} := infix.$

The above solution is in fact more general than what is needed for the purpose of embedding a new core language into the HPGP functor language, in that it makes the core language grammar more or less independent of the functor language extension. For embedding a new core language into the HPGP framework, one anyways has to develop a grammar and implement a parser for this language tailored to the needs of the functor language. Thus it is no problem if the resulting core language grammar is functor language dependent (the Cora grammar and parser are indeed functor language dependent, cf. Section 3.4 and Appendix A).

As a final remark, note that our syntax domain parameters are something like "global syntax domain variables". One could also think of really parameterized syntax domains $domain(parameter)$, which would allow different instances (or "applications") like $domain(domain_1)$ and $domain(domain_2)$ of such domains to appear in the same grammar. However, this was not necessary for the purpose of integrating different core languages into the Mona parser, since the necessary domain parameter settings are fixed for any kind of Mona translation unit, i.e., can be done as soon as the first keyword (`specification`, `functor` or `module`) of a source file is known.

3.2 Parameterized BISON Parsers

Now we show how BISON can be used to develop *parameterized parsers* for the parameterized grammars described in the previous section, i.e., parsers getting other parsers (for the unspecified syntax domains of the grammar) as parameters. We start with a closer look at the structure of a BISON-generated parser, in particular concerning the communication with the corresponding scanner, and describe then our modifications of the BISON output and how the result can be used to implement cooperating parsers.

3.2.1 Communication between BISON Parsers and Scanners

We assume that the reader is familiar with the principles of using BISON and FLEX (or YACC and LEX, for that purpose; see, e.g., [LMB92] for an introduction). To have something concrete to talk about, Program 3.1 on page 24 shows a simple BISON specification of a parser for propositional logic formulae (compare the grammar part of the BISON specification with the first core language grammar on page 20). The actions of the parser specification generate syntax trees of the parsed constructs. After having successfully parsed a complete formula, its syntax tree is assigned to the global variable `syntaxTree`.

A FLEX specification of a corresponding scanner is given in Program 3.2 on page 25. FLEX generates from this a definition of the function `yylex`, which scans the input for the next token and returns its token number. In case of an identifier `ID`, the corresponding *attribute* is assigned to the variable `yylval`. The definitions of the token numbers, as well as the declaration of `yylval`, are contained in the header file `y.tab.h`, which is generated by BISON from the `%token` (or `%left` and `%right`) declarations and the `%union` declaration in the parser specification.

BISON generates a definition of the parse function `yyparse` from a template with roughly the following structure:

```
int     yychar;         // the lookahead token
YYSTYPE yylval;         //   and its attribute
...

int yyparse() {         // the parser
    ...
    YYSTYPE yyval;      // this is the "$$" of the actions

    yychar = YYEMPTY;  // no lookahead token available

  yynewstate:
    ...
    if (yychar == YYEMPTY) yychar = yylex();

    // SHIFT or REDUCE

    goto yynewstate;
}
```

The type `YYSTYPE` is defined as the C `union` corresponding to the `%union` declaration (or as `int` if no `%union` is given). The function `yyparse` contains local definitions of the parser stacks (a stack of parser *states*, and a stack of the corresponding *attributes*), as well as of the variable `yyval`, which can be accessed in the actions of the parser specification as "$$" ("$1", "$2", ... refer to the attribute stack mentioned before).

In `yyparse`, first the token number `yychar` of the current lookahead token is set to the special value `YYEMPTY`, which indicates that in fact no lookahead token is

```
%{
#include "Syntax.h"  // syntax trees

extern int  yylex  ();
extern void yyerror(char *);

Syntax::Tree syntaxTree;
%}

%union {
  Syntax::Tree tree;
}

%right        IMPL
%left         OR
%left         AND

%token        NOT

%token<tree>  ID

%type<tree>   form infix prefix

//-----------------------------------------------------------
%%
form:  infix       { syntaxTree = $1; };

infix:
  prefix
| infix IMPL infix  { $$ = Syntax::impl($1, $3); }
| infix OR   infix  { $$ = Syntax::disj($1, $3); }
| infix AND  infix  { $$ = Syntax::conj($1, $3); }
;

prefix:
  ID
| NOT prefix        { $$ = Syntax::neg($2); }
| '(' infix ')'     { $$ = $2; }
;
```

Program 3.1: BISON Parser for Propositional Logic Formulae

```
%{
#include "Syntax.h"     // syntax trees
#include "y.tab.h"      // bison generated definitions

static int lineno = 1;  // current input line number
%}
//-----------------------------------------------------------
%%
"=>"    return IMPL;
"||"    return OR;
"&&"    return AND;
"!"     return NOT;

[a-zA-Z][a-zA-Z0-9]*  {
        yylval.tree = Syntax::id(yytext);
        return ID;
       }

[ \t]+  ;
\n      lineno++;

.       return yytext[0];
%%
//-----------------------------------------------------------

void yyerror(char *msg) {
  fprintf(stderr, "line %d: %s at <%s>\n",
                  lineno, msg, yytext);
};
```

Program 3.2: FLEX Scanner for Propositional Logic Formulae

available, i.e., if one is needed, the scanner has to be called. Then, in a loop until a sentence corresponding to the given grammar was successfully parsed, either the lookahead token is *shifted*, or the right-hand side of a grammar rule at the end of the parser stack is *reduced* to the left-hand side nonterminal of the rule. What to do (whether to shift or to reduce, and, in case of a reduction, which rule to use) depends on the current state and possibly the lookahead token. The decision is made by consulting a bunch of tables generated by BISON from the grammar part of the parser specification. If for this decision a lookahead token `yychar` is needed (reductions may be possible without lookahead) but not yet available, the scanner `yylex` is called before.

In case of a *shift*, the new state corresponding to the lookahead token `yychar` is pushed on the state stack, the attribute `yylval` of the lookahead token is pushed on the attribute stack, and `yychar` is reset to `YYEMPTY`.

For a *reduction* corresponding to some rule, first `yyval`, the "$$" value of the rule, is set to its default value ("$1") from the attribute stack. Then the corresponding action code from the grammar rule is executed. BISON collects all actions from the grammar part of the specification in a switch statement, with each action forming one case, and inserts this into the code of the template parser. Finally, the parts of the two parser stacks corresponding to the right-hand side of the reduced rule are popped, and the corresponding new state, as well as the attribute of the left-hand side nonterminal (i.e., the "$$" value yyval), is pushed. For more details about the inner workings of a parser generator like BISON or YACC (as well as a scanner generator like FLEX or LEX) we refer the reader to [ASU86].

3.2.2 Changing the BISON Parser Template

The essential point in making several BISON parsers cooperate is to communicate the lookahead tokens—one parser might have needed a lookahead just to decide that another parser has to be called, which then needs the same lookahead token as first token to shift.

The easiest way to solve this is to define only one variable yychar for the lookahead token (and analogously for its attribute yylval), externally to and commonly used by all cooperating parsers. Since the definition of yychar and yylval is hard-coded into the template parser used by BISON, we modify this template as follows (compare with the original structure shown in the previous subsection):

```
extern int      yychar;  // the lookahead token
extern YYSTYPE yylval;  //   and its attribute
extern YYSTYPE yyval;   // this is the "$$" of the actions
...

int yyparse() {         // the parser
    ...

  yynewstate:
    ...
    if (yychar == YYEMPTY) yylex();

    // SHIFT or REDUCE

    goto yynewstate;
}
```

Note that yychar is no longer initialized to YYEMPTY, since it might already contain a lookahead token read by the calling parser. Thus the initialization has to be done externally, before calling the top level parser.

Furthermore, note that we made also yyval external. This makes explicit communication of the attributes of the top level nonterminals between cooperating parsers unnecessary: after the call yyparse() of a parser for some syntax domain dom, the "$$" of dom is already correctly set for the calling parser, whose grammar contains dom as domain parameter. For the core language parser given in Program 3.1, this means that the variable `syntaxTree` and the assignment to it in the rule for the top level domain `form` is unnecessary, because the syntax tree of the parsed formula is available in the global variable `yyval` after execution of the parser.

Finally, since the variable `yychar` is now global and hence also available to the scanner, we use a scanner that directly assigns the number of the read token to `yychar` instead of returning it. The call of the scanner in the parser template is changed accordingly to just `yylex()` instead of `yychar = yylex()`.

Now, all the external variables have to be defined somewhere, i.e., in the scanner. We use only a single scanner, recognizing all tokens needed in any of the cooperating parsers. On the one hand, it is rather hard to use different scanners specialized to syntactic sub-domains, because problems arise when tokens that have to be communicated between parsers do not occur in both involved grammars: the calling parser might need a token not occurring in its grammar as lookahead to decide that the other parser has to be called, but the corresponding scanner recognizes, e.g., an identifier instead of a keyword, or only the "=" of the operator "=>". On the other hand, it is usually not a big deal to combine the FLEX specifications of a core language and an extension language scanner.

A FLEX specification for a scanner for predicate logic formulae according to the new conventions is shown in Program 3.3 on page 28 (compare it with the scanner specification from the previous subsection given in Program 3.2). The C preprocessor macro `YY_DECL` gives the prototype of the scan function produced by FLEX—its default definition is "`int yylex()`".

The BISON `%token` and `%union` declarations needed to generate the header file `y.tab.h` included in the scanner are:

```
%{
#include "Syntax.h"  // syntax trees
%}

%union {
  Syntax::Tree tree;
  Syntax::Seq  seq;  // sequence of syntax trees
}

// tokens for quantifiers:
//------------------------
%token        FORALL EXISTS

%token<tree>  ID

// additional core language tokens:
//---------------------------------
%right        IMPL
%left         OR
%left         AND

%token        NOT
```

```
%{
#include "Syntax.h"     // syntax trees
#include "y.tab.h"      // bison generated definitions

#define YY_DECL void yylex()  // scanner interface

int     yychar;         // current token
YYSTYPE yylval;         // attribute of current token
YYSTYPE yyval;          // attribute of current nonterminal

static int lineno = 1;  // current input line number
%}
//------------------------------------------------------------
%%
  // scanner rules for quantifiers:
  //------------------------------
forall  { yychar = FORALL; return; }
exists  { yychar = EXISTS; return; }

[a-zA-Z][a-zA-Z0-9]*  {
          yychar = ID;
          yylval.tree = Syntax::id(yytext);
          return;
         }

  // additional core language rules:
  //-------------------------------
"=>"    { yychar = IMPL;   return; }
"||"    { yychar = OR;     return; }
"&&"    { yychar = AND;    return; }
"!"     { yychar = NOT;    return; }

  // general rules:
  //---------------
[ \t]+    ;
\n        lineno++;

.       { yychar = yytext[0]; return; }
%%
//------------------------------------------------------------

void yyerror(char *msg) {
  fprintf(stderr, "line %d: %s at <%s>\n",
                  lineno, msg, yytext);
};
```

Program 3.3: FLEX Scanner for Predicate Logic Formulae

```
%{
extern void yylex  ();
extern void yyerror(char *);

#define yyparse   Quan_parse
#define yyparse_P Quan_parse_P

typedef int (*Parser)();
Parser yyparse_P;
%}

%type<tree>  form quan _P_
%type<seq>   ids

//----------------------------------------------------------
%%
form:  quan               { YYACCEPT; };

quan:
  _P_
| FORALL '(' ids ')' quan  { $$ = Syntax::univ($3, $5); }
| EXISTS '(' ids ')' quan  { $$ = Syntax::exis($3, $5); }
;

ids:
  ID                       { $$ = Syntax::seq($1);     }
| ids ',' ID               { $$ = Syntax::app($1, $3); }
;

_P_:                   { if (yyparse_P() != 0) YYERROR; };
```

Program 3.4: BISON Parser for Quantified Formulae

These declarations have to be prepended to all the parser specifications given in the following, so that all cooperating parsers understand the same token numbers and use the same C type YYSTYPE for attribute values.

To show how the modified BISON parsers can be used, Program 3.4 contains a BISON specification of a parser for quantified formulae parameterized with core language parsers for propositional logic formulae (compare this with the parameterized grammar rule for the syntax domain *quan* given in the previous section on page 21).

In the initial part of this specification, the parser function yyparse generated by BISON is renamed to Quan_parse with a C preprocessor #define, so that different cooperating parsers get different names.

The parser Quan_parse_P for the syntax domain parameter _P_ is globally

```
%{
extern void yylex  ();
extern void yyerror(char *);

#define yyparse   Infix_parse
#define yyparse_P Infix_parse_P

typedef int (*Parser)();
Parser yyparse_P;
%}

%type<tree>  form infix _P_

//-----------------------------------------------------------
%%
form:  infix        { YYACCEPT; };

infix:
  _P_
| infix IMPL infix  { $$ = Syntax::impl($1, $3); }
| infix OR   infix  { $$ = Syntax::disj($1, $3); }
| infix AND  infix  { $$ = Syntax::conj($1, $3); }
;

_P_:            { if (yyparse_P() != 0) YYERROR; };
```

Program 3.5: BISON Parser for Infix Formulae

defined as C function pointer, which has to be set to some core language parser from outside, before `Quan_parse` can be used. `Quan_parse_P` is called in the action of the grammar rule for `_P_`, whose right-hand side is otherwise empty. A non-zero return value of the called parser indicates that a syntax error was detected—in this case, the internal error handling of the calling parser is triggered by executing the special action `YYERROR`.

The rule for the top level domain `form` contains the special action `YYACCEPT` telling the parser to immediately return with success (return value zero) at this point, i.e., as soon as a reduction corresponding to this rule has happend. This is necessary if phrases of the top level domain do not necessarily comprise whole input files, since if no `YYACCEPT` is executed before, the BISON-generated parser will indicate a syntax error if anything else than the end of the input file comes after the parsed construct.

Programs 3.5 and 3.6 show specifications for cooperating parsers recognizing propositional logic formulae (compare with the grammar on page 21).

A top level parser `Form_parse` for formulae can be defined using another parser

```
%{
extern void yylex  ();
extern void yyerror(char *);

#define yyparse   Prefix_parse
#define yyparse_A Prefix_parse_A
#define yyparse_P Prefix_parse_P
#define yyparse_F Prefix_parse_F

typedef int (*Parser)();
Parser yyparse_A, yyparse_P, yyparse_F;
%}

%type<tree>  form prefix _A_ _P_ _F_

//-----------------------------------------------------------
%%
form:  prefix  { YYACCEPT; };

prefix:
  _A_
| NOT _P_      { $$ = Syntax::neg($2); }
| '(' _F_ ')'  { $$ = $2; }
;

_A_:  { if (yyparse_A() != 0) YYERROR; };
_P_:  { if (yyparse_P() != 0) YYERROR; };
_F_:  { if (yyparse_F() != 0) YYERROR; };
```

Program 3.6: BISON Parser for Prefix Formulae

parameter `Form_parse_F` as follows:

```
typedef int (*Parser)();
Parser Form_parse_F;

int Form_parse() {

  yychar = YYEMPTY;

  return Form_parse_F();
}
```

Note the initialization of `yychar`—the definition should appear at some place where `YYEMPTY` is defined, e.g. at the end of one of the above BISON parser

```
%{
extern void yylex  ();
extern void yyerror(char *);

#define yyparse Predicate_parse
%}

%type<tree>  form predicate
%type<seq>   args ids

//-----------------------------------------------------------
%%
form:  predicate   { YYACCEPT; };

predicate:
  ID '(' args ')'  { $$ = Syntax::pred($1, $3); };

args:
  /* empty */      { $$ = Syntax::seq(); }
| ids
;

ids:
  ID               { $$ = Syntax::seq($1);     }
| ids ',' ID       { $$ = Syntax::app($1, $3); }
;
```

Program 3.7: BISON Parser for Predicates

specifications. This parser can be set up to parse predicate logic formulae with the following assignments:

```
Form_parse_F   = Infix_parse;

Infix_parse_P  = Quan_parse;

Quan_parse_P   = Prefix_parse;

Prefix_parse_A = Predicate_parse;
Prefix_parse_P = Quan_parse;
Prefix_parse_F = Infix_parse;
```

Thereby, `Predicate_parse` must be a parser for the syntax domain $predicate$ from the previous section, as for instance the one specified in Program 3.7.

The same parser `Form_parse` can, however, also be used to parse propositional logic formulae, namely using the set-up:

```
Form_parse_F   = Infix_parse;

Infix_parse_P  = Prefix_parse;

Prefix_parse_A = ID_parse;
Prefix_parse_P = Prefix_parse;
Prefix_parse_F = Infix_parse;
```

For this we need a parser ID_parse for a single identifier—we could of course give a BISON specification with only a single grammar rule like

```
%{
...
#define yyparse ID_parse
%}

%%
form:  ID  { YYACCEPT; };
```

but it is very easy to write such a simple parser by hand:

```
int ID_parse() {

  if (yychar == YYEMPTY) yylex();
  if (yychar != ID) {
    yyerror("parse error");

    return 1;
  }
  yyval  = yylval;
  yychar = YYEMPTY;

  return 0;
}
```

Again, YYEMPTY, and also the token numbers, must be known at the place where this definition appears.

3.2.3 Grammar Requirements

For implementing calls to sub-parsers by BISON grammar rules with empty right-hand sides, as described in the previous subsection, the used grammar must fulfill certain conditions: If, at a certain point during parsing, a sub-parser might be called, it must be impossible that any of the tokens accepted as first ones by the sub-parser are shifted instead. In other words, it must be completely clear from looking at at most one lookahead token that the sub-parser has to be called. Consider the following excerpt from a BISON specification of a tree-generating expression parser, which allows dereferencing in the form "*type* `@` *address*" like the Cora parser does:

```
prefixExpr:
  ...
| typeExpr '@' prefixExpr  { $$ = Syntax::deref($1, $3); }
| ...
| ID
;

typeExpr:  { if (TypeExpr_parse() != 0) YYERROR; };
```

Now consider the structure of the specification of `TypeExpr_parse`:

```
typeExpr:
  RECORD ...
| ID
;
```

The condition stated in the beginning means that, in the expression parser, none of the tokens `RECORD`, `UNION` or `ID` can be shifted instead of reducing corresponding to the rule for `typeExpr` and thus calling `TypeExpr_parse`. This is not the case: an `ID` can be shifted, which means that it *will* be shifted, and if an "`@`" follows and is not allowed there by some other rule of the expression grammar, the expression parser will return to its caller, where probably a syntax error is detected.

Rewriting the first rule of the expression parser specification as follows makes the expression parser work:

```
prefixExpr:
  ...
| typeExpr '@' prefixExpr  { $$ = Syntax::deref($1, $3); }
| ID       '@' prefixExpr  { $$ = Syntax::deref($1, $3); }
| ...
| ID
;
```

Note however that this requires knowledge of the type expression grammar by the developer of the expression parser (no problem in case of Mona: both are core language parsers), and that the above rule, if combined with the grammar from the type expression parser, is now ambiguous (but both ways of parsing "id `@` $address$" produce the same syntax tree).

3.2.4 Resolving Induced Parsing Conflicts

The described approach for implementing parameterized parsers works due to the fact that BISON generates LALR(1) parsers, which usually continue to do some reductions if possible, also if it is already clear from the grammar and the lookahead token that there must be a syntax error. Take, e.g., the grammar part of the parser for quantified formulae from above (Program 3.4 on page 29). It seems that the first read token can only be one of the quantifiers `FORALL` or `EXISTS`, since the right-hand side of `_P_`, the third alternative of `quan`, looks like allowing only the empty word. An LR(1) parser would immediately stop and complain about a syntax error if, e.g., an identifier would appear as first token, whereas the BISON-generated LALR(1) parser tries to reduce the rule for `_P_`, which results in a call of the `Quan_parse_P` parser—the called parser then has to decide whether an identifier is allowed at this place (or it calls again another parser to make the decision...).

Still, also with BISON some problems are likely to appear with the empty right-hand sides of the rules for syntax domain parameters. Consider, e.g., the following excerpt from the specification of a split expression parser, fixing the syntax of function applications:

```
postfixExpr:
    ...
  | postfixExpr '(' tuple ')'
    ...

tuple:
    /* empty */
  | exprs
  ;

exprs:
    _E_
  | exprs ',' _E_
  ;

_E_:  { if (yyparse_E() != 0) YYERROR; };
```

BISON complains about a *reduce/reduce conflict*: if, after having reduced a `postfixExpr`, and having then shifted an opening parenthesis "(", anything other than a comma "," is the lookahead token, it is not clear whether to reduce corresponding to the empty alternative of the rule for `tuple`, or corresponding to the rule for `_E_` (such that the `_E_` can later on be reduced to `exprs`, and this, in turn, to a `tuple`). BISON resolves this per default by reducing corresponding to the rule appearing first in the parser specification. In the above case, this is the rule for `tuple`, which is wrong except if the lookahead token is a closing parenthesis ")".

In this case, a grammar transformation helps: rewriting the first two rules of the above grammar part to

```
postfixExpr:
    ...
  | postfixExpr tuple
    ...

tuple:
    '(' ')'
  | '(' exprs ')'
  ;
```

partially resolves the reduce/reduce conflict. A *shift/reduce conflict* remains: if, after having shifted the opening parenthesis "(", the next lookahead token is the closing parenthesis ")", it is not clear whether to shift the lookahead token (so that next the two parentheses can be reduced corresponding to the first alternative of the rule for `tuple`), or to first reduce the empty word corresponding to the rule for `_E_` (so that next a reduction corresponding to the first alternative of the rule for `exprs` can be done, and then a reduction corresponding to the second alternative of the rule for `tuple`). BISON chooses per default the shift, which is what we want in this case, provided that the parser supplied for the `yyparse_E` parser parameter does not accept a closing parenthesis ")" as first token, and of course not the empty word.

If one does not want to rely on BISON's default solution, the shift/reduce conflict can be resolved explicitly by choosing appropriate *precedences* for ")" and the rule for `_E_`. However, this may cause other troubles in some cases. Another possibility is to remove the first alternative of the rule for `tuple` and to hand-write the parsing of the `exprs` domain so that additionally the empty word is accepted. For this, one has to make the rule for `exprs` empty and do the parsing in its action code using techniques derived from the ones used in the following examples.

Another conflict results from optional occurrences of constructs parsed by external parsers. Consider, e.g., the `return` command with or without return value in combination with an external parser `Expr_parse` for expressions:

```
returnCmd:
  RETURN
| RETURN expr
;

expr:  { if (Expr_parse() != 0) YYERROR; };
```

This does not work, since after having shifted a `RETURN` token, it is not clear whether to reduce an `expr` or not (BISON will per default never try to parse an `expr`).

If the set of terminal symbols accepted as first tokens by `Expr_parse` is disjoint from the set of terminal symbols that can follow a command (and if `Expr_parse` does not accept the empty word, what would make the grammar ambiguous), this can be resolved by explicitly checking the next input token. Assume that a boolean function `isExprLA` is given, which decides whether an expression can start with the token passed as argument. Then the following would work:

```
returnCmd:  RETURN exprOpt;

exprOpt: {
  if (yychar == YYEMPTY) yylex();
  if (isExprLA(yychar)) {
    if (Expr_parse() != 0) YYERROR;
  }
};
```

A similar situation is at hand if at a certain point during parsing two different sub-parsers might be called. Consider, e.g., the following excerpt from a command parser specification:

```
cmd:
  cmdBlock
| ifCmd
| whileCmd
| returnCmd
| callCmd
| extCmd
;

cmdBlock:   '{' ...
ifCmd:      IF ...
whileCmd:   WHILE ...
returnCmd:  RETURN ...

callCmd:   { if (CallCmd_parse() != 0) YYERROR; };

extCmd:    { if (ExtCmd_parse()  != 0) YYERROR; };
```

Thereby, `CallCmd_parse` is a parser for procedure calls, and `ExtCmd_parse` is a parser for some command domain extension (e.g. constructor and destructor calls). If the first token is none of "{", `IF`, `WHILE` or `RETURN`, it is not clear which one of the two external parsers to call. If the set of terminal symbols accepted as first tokens by `ExtCmd_parse` is disjoint from the possible first tokens of all other alternatives of `cmd`, and analogously for `CallCmd_parse`, this can again be resolved by explicitly checking the next input token. Assume, e.g., that the first token of an `extCmd` must be either `CONSTR` or `DESTR`, and that none of "{", `IF`, `WHILE`, `RETURN`, `CONSTR` and `DESTR` can be the first token of a `callCmd`. Then one could write the command parser as:

```
cmd:
  cmdBlock
| ifCmd
| whileCmd
| returnCmd
| callCmd
;

...

callCmd: {
  if (yychar == YYEMPTY) yylex();
  if (yychar == CONSTR || yychar == DESTR) {
    if (ExtCmd_parse()  != 0) YYERROR; };
  } else {
    if (CallCmd_parse() != 0) YYERROR; };
  }
};
```

We assumed here that neither of the called parsers `ExtCmd_parse` or `CallCmd_parse` accepts the empty word. If one of them does, the tokens that can follow a command have to be added to the set of terminal symbols accepted as first tokens by this parser, and the resulting set must be disjoint from the set of possible first tokens of all other command alternatives. This implies that if both external parsers accept the empty word, we are out of luck. Similarly, if one of the other alternatives of `cmd` may be empty—which is obviously not the case in the above example—the tokens that can follow a `cmd` have to be treated like possible first tokens of `cmd`.

3.2.5 Managing Token Definitions

One more problem with the described scheme for implementing parameterized parsers are the token definitions `%token`, `%left` and `%right`, as well as the token attribute type declaration `%union`, which have to be the same for all involved sub-parsers, and must coincide with what the scanner uses. To avoid having to maintain multiple copies of these definitions, we wrote wrapper shell scripts for BISON and FLEX, which allow to use one single token definition file separate from the parser and scanner specifications, and also do all the other required transformations of the produced parser and scanner C sources.

The script `subbison` produces a BISON parser from a corresponding specification, and modifies the contained parser template in the described way. It understands all BISON options. Additionally, a file *token-defs* containing `%token`, `%left` and `%right` declarations and a `%union` declaration (like shown on page 27) can be given with the option "`-u`*token-defs*". This file is prepended to the parser specification before BISON is called, allowing to keep only one copy of the token and attribute type definitions for all cooperating parsers.

The script `multiflex` produces a FLEX scanner from a corresponding specification, with additional global definitions of the variables `yychar`, `yylval` and `yyval`, as well as the definition of `yylex` as procedure (instead of as function returning `int`) via the macro `YY_DECL`, so that these do not have to be given explicitly in the scanner specification (like in Program 3.3 on page 28). All FLEX options are understood, as well as the option "`-u`*token-defs*", with which the C definitions corresponding to the `%token`, `%left`, `%right` and `%union` declarations contained in the file *token-defs* are included directly in the generated scanner, so that no `y.tab.h` header has to be included in the specification.

3.3 Syntax Error Recovery

Error handling in BISON is done with the special `error` token. If a syntax error is detected, first the procedure `yyerror` is called with an error message (usually just "`parse error`") as argument to print an error message. Then the parser stacks are popped until an `error` token can be pushed because of some grammar rule containing `error` in its right-hand side. If no such rule can be found, the generated parser returns a non-zero value to its caller as soon as the stacks are empty.

If, in a found `error` rule, the `error` token is not the last right-hand side token, the generated parser reads input tokens until a token is found that can follow the `error` in this rule, and then continues normal parsing. In fact, the parser considers itself as having recovered from the error only after a certain number of tokens was successfully pushed to the stack; if another error occurs before, it produces no error message before again starting to pop the stacks.

If the `error` token is the last right-hand side token of the `error` rule, this rule is reduced immediately, executing the corresponding action, before the parser starts to scan the input for a token that can follow the `error`. This allows to handle errors by hand in the action code of the error rule. One can, e.g., try to find an input token at which parsing should continue. The special action `yyerrok` can then be used to tell the parser that recovery is complete, so that normal parsing is continued immediately after the `error` rule reduction.

Having parsers for sub-domains that can be called as procedures provides now some added flexibility for doing the error recovery by hand in the actions to `error` rules. Consider, e.g., a function definition in a Mona functor implementation: If a syntax error is detected in the head of the definition, it is possible to look for a possible first token of the function body, and, if one is found, to call the function body parser. If none is found, e.g. because the function definition is in fact only a `forward` declaration, parsing can be continued at the next input token that can follow a function definition or after a terminating semicolon "`;`" was found. The above scenario requires of course knowledge of possible first tokens of the core language function body domain in the functor language definition parser. We manage this by using certain boolean functions in the functor language parsers. These functions decide for certain core language domains whether these can start with the token passed as argument. They have to be implemented for any core language to be embedded.

Using external parsers with the techniques from the previous section poses, however, also a problem with BISON error recovery: If, at some point during parsing where an external parser should be called (i.e., the empty right-hand side of some rule reduced), also an `error` token can be shifted, the BISON-generated parser will in any case do the shift and start error handling, and never do the reduction calling the external parser. So one has to be careful with writing `error` rules (BISON reports no parsing conflict for such situations).

3.4 Structure of the Mona Parser

In the following we describe the actual structure of the Mona parser, i.e., what sub-parsers are used for parsing the functor language and core language parts of Mona, and how the set-up of these sub-parsers is done, i.e., how the parser parameters are set.

For each parser listed in the following, the syntax domain accepted by the provided `parse` function is given (the domain names refer to the grammar listed in Appendix A), as well as the fixed sub-parsers that are used, and the parser variables, which have to be set before usage (together with the syntax domain they are expected to accept).

There are some small syntax domains, like specification expressions $specExpr$, module expressions $moduleExpr$, or qualified identifiers $longId$ and $globalId$, which are used in several of the sub-parsers, but for which currently no separate parsers are provided—their definition is duplicated where it is needed. We intend to provide hand-written parsers for these domains as soon as the HPGP project moves on to higher order functors.

3.4.1 Functor Language

The top level parser accepting Mona translation units is:

`Unit`

Accepts: Mona translation units $unit$.

Uses: `Spec`, `Def`, `RestrDef`.

The following parsers are used to parse the functor language parts of Mona:

`Spec`

Accepts: specifications $spec$.

Requires to be set:

`parse_TypeExpr` accepting type expressions $typeExpr$,

`parse_ValueExpr` accepting value expressions $valueExpr$.

`Def`

Accepts: definitions def.

Requires to be set:

`parse_Cmd` accepting commands cmd,

`parse_TypeExpr` accepting type expressions $typeExpr$.

`RestrDef`

Accepts: restricted definitions $restrDef$ in `module` units.

A separate `RestrDef` parser is not necessary for keeping the functor language parser independent of the core language, but provided to be called from the error recovery code in the `Unit` parser.

The following parsers are provided to be called from core language parsers:

`LocalDef`

Accepts: local definitions in command blocks $localDef$.

Requires to be set:

`parse_TypeExpr` accepting type expressions $typeExpr$,

`parse_ValueExpr` accepting value expressions $valueExpr$.

`ExtCmd`

Accepts: constructor and destructor calls $extCmd$.

Requires to be set:

`parse_TypeExpr` accepting type expressions $typeExpr$,

`parse_ValueExpr` accepting value expressions $valueExpr$.

`AxiomExpr`

Accepts: value expression extensions in `specification` units $axiomExpr$.

Uses: `Spec`.

Requires to be set:

`parse_TypeExpr` accepting type expressions $typeExpr$,

`parse_PrefixExpr` accepting prefix value expressions $prefixExpr$.

`ReprExpr`

Accepts: value expression extensions in `functor` units $reprExpr$.

Requires to be set:

`parse_TypeExpr` accepting type expressions $typeExpr$,

`parse_PrefixExpr` accepting prefix value expressions $prefixExpr$.

3.4.2 Core Language

The following parsers are used to parse the core language parts of Mona:

`Cmd`

Accepts: commands cmd.

Uses: `LocalDef`, `ExtCmd`, `Opt`, `ValueExpr`.

`Opt`

Accepts: `switch` options opt.

Uses: `Cmd`, `ValueExpr`.

`TypeExpr`

Accepts: type expressions $typeExpr$.

`ValueExpr`

Accepts: value expressions $valueExpr$.

Uses: `PrefixExpr`.

`PrefixExpr`

Accepts: prefix value expressions $prefixExpr$.

Uses: `TypeExpr`, `ValueExpr`.

Requires to be set:

`parse_ExtPrefixExpr` accepting value expression extensions $extPrefixExpr$.

A separate `Opt` parser is not required by the functor language, but provided to be called from the error recovery code in the `Cmd` parser.

3.4.3 Parser Set-up

The required parser variables are set in the `Unit` parser as soon as it is clear which kind of unit to parse. In case of a `specification` unit, the following assignments are executed immediately before `Spec::parse()` is called:

```
Spec      ::parse_TypeExpr      = TypeExpr  ::parse;
Spec      ::parse_ValueExpr     = ValueExpr ::parse;

PrefixExpr::parse_ExtPrefixExpr = AxiomExpr ::parse;

AxiomExpr ::parse_TypeExpr      = TypeExpr  ::parse;
AxiomExpr ::parse_PrefixExpr    = PrefixExpr::parse;
```

In case of a `functor` unit, the following assignments are executed immediately before `Def::parse()` is called:

```
Def       ::parse_Cmd           = Cmd       ::parse;
Def       ::parse_TypeExpr      = TypeExpr  ::parse;

LocalDef  ::parse_TypeExpr      = TypeExpr  ::parse;
LocalDef  ::parse_ValueExpr     = ValueExpr ::parse;

ExtCmd    ::parse_TypeExpr      = TypeExpr  ::parse;
ExtCmd    ::parse_ValueExpr     = ValueExpr ::parse;

PrefixExpr::parse_ExtPrefixExpr = ReprExpr  ::parse;

ReprExpr  ::parse_TypeExpr      = TypeExpr  ::parse;
ReprExpr  ::parse_PrefixExpr    = PrefixExpr::parse;
```

In case of a `module` unit, no set-up is necessary, since no core language parsers are involved.

Chapter 4

Special Parser Features

Some of the special features of Mona are already completely resolved by the parser. They are thus not at all visible to the compiler modules processing the syntax tree generated by the parser. These modules (most notably the type checker) are hence relieved from the necessity of implementing some special treatment.

This chapter describes how the parser handles these features, which are:

- user-definable operators,
- user-definable character, string and number literals, and
- a restricted set of identifiers for items referring to files, so that these identifiers can be used directly as file names.

The first two are pure core language features, whereas the third one is provided by the functor language—it is discussed first, together with a description of the different kinds of identifiers used in Mona.

4.1 Identifiers

The Mona scanner provides a terminal class `unitId` for simple identifiers consisting of letters and digits, with a letter as first character (cf. Appendix B). These are used to identify the globally visible Mona specifications, functors, modules, or packages. These items are represented by certain files in a central repository, and the simple identifiers serve for a simple mapping to file path names.

Mona units, i.e., specifications, functors and modules, may be 'contained' in packages, in which case it is necessary to qualify the unit name with the package to refer to the unit. The resulting package-qualified identifiers are called *global identifiers* $globalId$, and have the form

$[packageExpr$`<>`$]$`unitId`

with the part in brackets being optional, for units not contained in a package. Since packages can themselves be contained in other packages, so that a hierarchical package structure can be built, a $packageExpr$ designating a certain package in the hierarchy has again the form of a global identifier. This results in identifiers of the form `P<>Q<>...<>U` for Mona units and packages, where all components `P`, `Q`,..., `U` are of the restricted ***unitId*** form.

However, for items 'inside' Mona units, i.e., for the modules, types, procedures, functions, constants and variables specified in `specification` units and defined in `functor` units, as well as for the module parameters of these units, and for the modules defined in `let` blocks in `module` units, the Mona parser allows identifiers id with extended lexical syntax. An id has either the form of a ***unitId*** as defined by the functor language, or of a ***coreId***, a terminal class defined in the core language part of the Mona scanner. This can be used to provide a much richer set of identifiers, containing all kinds of special characters, from the core language side. In Cora, this is used on the one hand to allow underscore characters "_" in identifiers, as is usual in C or C++, and on the other hand to provide support for user-definable operators and literals as described in the following two sections.

To refer to items exported by a module, it is necessary to qualify the item identifiers with the module (if the module is not imported). The resulting module-qualified identifiers are called *long identifiers* $longId$, and have the form

$$[moduleExpr\texttt{::}]id$$

with the part in brackets again being optional, for items in the current unit. A $moduleExpr$ determines a module—it can either be a module identifier $moduleId$, or the application of a functor, denoted by a $functorExpr$, to a list of argument modules. The allowed form of a $moduleId$, as well as of a $functorExpr$, depends on the kind of unit in which it occurs:

- In `specification` and `functor` units, it is not possible to refer to globally visible modules, but only to module parameters of, and modules specified or defined in, the unit, and modules exported by these. A $moduleId$ can hence only be a $longId$. This results in identifiers of the form `M::N::...::I`, where all components `M`, `N`,..., `I` can be of the general id form.

  Access to globally visible functors is restricted to *functor usage declarations* $functorUse$ in `specification` and `functor` units. Such a declaration associates a global functor with a local name id, which can be used later on to denote the functor in functor applications. Since this local name is not exported (the HPGP system is not yet in 'higher order' state), a simple id is the only possible form of a $functorExpr$. This results in long identifiers of the form `F[...]::N::...::I`, where all components `F`, `N`,..., `I` can be of the general id form.

- In `module` units, on the other hand, all global items can be used. Thus, a module identifier $moduleId$ can either be a $longId$ like above, or as well a $globalId$. The latter results in identifiers of the form `P<>...<>M::N::...::I`, where the components `P`,..., `M` are of the restricted ***unitId*** form, whereas the components `N`,..., `I` can be of the general id form.

> Since functors cannot have local names in `module` units (the only definitions possible there are module definitions), a *functorExpr* can only have the form of a *globalId*, resulting in identifiers of the form `P<>...<>F[...]::N::...::I`, where the components `P`,..., `F` are of the restricted `unitId` form, whereas the components `N`,..., `I` can be of the general *id* form.

The distinction between the restricted `unitId` identifiers of the functor language and the extended `coreId` identifiers of the core language is only used by the parser to guarantee an easy mapping to file names where it is needed. The syntax tree for both identifier kinds is just an `id` leaf node, which refers to an entry in the identifier table—the difference between the two identifier kinds is then only visible from the textual representations of the identifiers, which are stored in the table.

4.2 Operators

The Cora parser allows user-defined binary and unary operators by allowing special *operator names* as identifiers, and by producing the same syntax trees for occurrences of the user-definable operators as for simple applications of the corresponding operator names. This is similar to what is provided by C++, except that there operator names can only be defined as functions with certain interfaces.

The user-definable binary infix operators are:

> "=", "|", "^", "&", "==", "!=", "<=", ">=", "<", ">", "<<", ">>", "+", "–", "*", "/" and "%".

The user-definable unary prefix operators are:

> "!", "~", "+", "–", "*" and "&".

Moreover, the two unary postfix operators "`.`*id*" and "`->`*id*" are user-definable. In fact, these are classes of operators: for any valid identifier *id* the two listed operators can be defined and used in postfix notation. And finally, also the binary indexing operator "... `[...]`" is user-definable.

The corresponding operator names are of the form "`operator` *op*", where *op* is one of the operators listed above, or "`[]`" for the indexing operator. There are a few more operator names allowed, cf. Appendix B. These are, however, used for implementing user-definable literals, and are thus listed in the following section.

The operator names are used by the parser for generating the syntax trees for expressions built with the above operators:

- For a binary infix expression "$expr_1\ op\ expr_2$", the same syntax tree is generated as for "`operator` op`(`$expr_1$`,` $expr_2$`)`".
- For a unary prefix expression "$op\ expr$", the same syntax tree is generated as for "`operator` op`(`$expr$`)`".

- For a unary postfix expression "$expr$`.`id", the same syntax tree is generated as for "`operator .`id`(`$expr$`)`". Analogously for "$expr$`->`id", with "`operator .`id" replaced by "`operator ->`id".

- Finally, for a binary indexing expression "$expr_1$`[`$expr_2$`]`", the same syntax tree is generated as for "`operator []`(`$expr_1$`,` $expr_2$`)`".

Now, since the operator names are treated as any other identifier when appearing in a Mona source file, any of the above operators can be defined by simply giving a procedure or function definition for the corresponding operator name. Thus, we have made operators user-definable in a way that is totally transparent for the type checker, which just has to treat usual procedure or function definitions and applications.

As an easy syntactical extension (which is still totally transparent for compiler modules processing the syntax tree), the Cora value expression parser allows also to qualify the above user-definable operators with modules $moduleExpr$. This allows to use operator notation for operators defined in other modules without having to import the corresponding operator names:

- Instead of any of the infix or prefix operators "op" listed above, "op`[`$moduleExpr$`]`" can be used as operator in infix, respectively prefix, notation.

- Instead of a unary postfix operator "`.`id", "`.[`$moduleExpr$`]`id" can be used as operator in postfix notation. Analogously for "`->[`$moduleExpr$`]`id" instead of "`->`id".

- And finally: instead of the indexing operator "...`[`...`]`", "...`[[`$moduleExpr$`]`...`]`" can be used.

In any of these cases, the parser produces the same syntax tree as for the unqualified operator, but with the node for the applied "`operator` op" identifier replaced by the tree for the corresponding long identifier "$moduleExpr$`::operator` op".

4.3 Literals

Character literals, as used in C, are fixed length items, and their number is thus finite. Therefore, they are easily made user-definable by considering them as identifiers (*character names*)—the finitely many characters can then be defined with finitely many constant definitions. This means that an item like "`'c'`" appearing in a source file, which is considered as a literal in C, is in Cora just a normal identifier (cf. Appendix B).

The other kinds of literals allowed in Cora (character strings and numbers) are, similar to operator expressions, just considered as special syntax for (nested) function applications, so that only the parser has to know about something like literals. The exact form of the function applications chosen in Cora to represent literals, as well as the special identifiers from which they are built, are described in detail in the subsections below.

Since string and number literals have to be recognized in the scanner (which knows about white space in the input, whereas the parser does not), the syntax trees for literals are already constructed by the Cora part of the scanner. They are communicated to the parser as attributes of the `lit` terminal class tokens.

Analogously as for operators, Cora allows to qualify also literals `lit` with modules $moduleExpr$, using the special syntax "$moduleExpr$`::lit`". This is resolved by the parser by replacing the identifier nodes for all identifiers "`id`" in the scanner-generated tree for "`lit`" by the syntax trees for the corresponding long identifiers "$moduleExpr$`::`id". This, now, has to be done by the parser, since the structure of a $moduleExpr$ is already too complex to be recognized by the scanner. Note that for character 'literals', module qualification yields, also syntactically, just a normal long identifier $longId$).

4.3.1 Character Strings

Character strings are translated by the Cora scanner into syntax trees for nested function applications built up with the following special identifiers:

- the character names of the single characters that make up the string,
- the *empty string marker* "`''`", and
- the operator name "`operator "`".

The syntax tree generated by the scanner for a string literal "`"abc"`" is the same as the one for the expression

```
operator "(
    operator "(
        'a',
        operator "(
            'b',
            operator "(
                'c',
                ''
)   )   )   )
```

This is a list-like structure, with the binary "`operator "`" representing the 'cons' operation, and "`''`" representing the empty list. The additional top-level application of a unary "`operator "`" allows, e.g., efficient translation of a character list structure produced by the binary "`operator "`" functions into a character array). The empty string "`""`" consequently corresponds to "`operator "('')`".

Just to demonstrate it once: If a string literal occurs qualified, e.g. as "`M::"abc"`", the above syntax tree generated by the scanner for "`"abc"`" is translated by the parser to the one for

```
M::operator "(
    M::operator "(
        M::'a',
        M::operator "(
            M::'b',
            M::operator "(
                M::'c',
                M::''
)   )   )   )
```

All identifiers occurring in the result are qualified with "M".

Using character string literals of some type String can now be made possible by providing definitions for the following items:

```
val '\0': Char
  ...
val '~':  Char

val '':                              CharList
fun operator "(c: Char, l: CharList): CharList

fun operator "(l: CharList): String
```

A character type Char and an intermediate character list type CharList can be chosen as needed. The following definitions, e.g., allow to use string literals of the built-in String type:

```
type Char builtin Char

val '\0': Char builtin '\0'
  ...
val '~':  Char builtin '~'

type String builtin String

val '':                            String builtin ''
fun operator "(c: Char, l: String): String builtin operator "

fun operator "(l: String): String builtin String
```

The built-in function "operator "" yields a built-in String consisting of the built-in Char argument prepended to the built-in String argument. The built-in function String is the identity on the built-in type String (identifiers for types reside in a separate name space, as do specification names, functor names and module names).

4.3.2 Integer Numbers

Cora allows integer numbers to be written in decimal or hexadecimal notation. They are translated by the scanner into syntax trees for nested function applications built up with the following special identifiers:

- the *digit names* of the single digits that make up the number, and
- the operator names "operator 0d" and "operator 0D" for decimal numbers, or
- the operator names "operator 0x" and "operator 0X" for hexadecimal numbers.

Digit names are the last kind of special identifiers allowed by the core language part of the scanner, just for the purpose of making number literals user-definable. They are of the form "#dig", where dig is a single decimal or hexadecimal digit, i.e., one of either "0",..., "9" or "A",..., "F".

The syntax tree generated by the scanner for an integer literal "123" (or "0d123") in decimal notation is the same as the one for the expression

```
operator 0d(
    operator 0D(
        operator 0D(
            operator 0D(#1),
            operator 0D(#2)
        ),
        operator 0D(#3)
)   )
```

This structure allows Horner-scheme evaluation: the binary function application "operator 0D(n, d)" could compute $10 * n + d$. A single digit literal "0" consequently corresponds to "operator 0d(operator0D(#0))".

The syntax tree generated by the scanner for an integer literal "0x9AB" in hexadecimal notation is the same as the one for the expression

```
operator 0x(
    operator 0X(
        operator 0X(
            operator 0X(#9),
            operator 0X(#A)
        ),
        operator 0X(#B)
)   )
```

Analogously to above, the binary function application "operator 0X(n, d)" could compute $16 * n + d$. A single digit literal "0xF" consequently corresponds to "operator 0x(operator0X(#F))".

Using integer literals of some type Int in decimal and hexadecimal notation is hence made possible by providing definitions for the following items:

```
val #0: Dig
  ...
val #F: Dig
```

```
fun operator 0D(d: Dig):         Dec
fun operator 0D(n: Dec, d: Dig): Dec

fun operator 0X(d: Dig):         Hex
fun operator 0X(n: Hex, d: Dig): Hex

fun operator 0d(n: Dec): Int
fun operator 0x(n: Hex): Int
```

Intermediate types `Dig`, `Dec` and `Hex` can be chosen as needed.

4.3.3 Floating Point Numbers

Floating point number literals are translated by the Cora part of the scanner into syntax trees for nested function applications structured like a combination of three integer literal trees, for the integer, fractional and exponent part. Thereby the following special identifiers are used:

- the digit names of the single digits that make up the number,
- the operator name "`operator 0I`" for the integer part,
- the operator name "`operator 0F`" for the fractional part,
- the operator names "`operator 1E`" and "`operator E-`" for the exponent part, and
- the operator name "`operator 0f`" for combining the three parts.

The syntax tree generated by the scanner for a floating point number literal "`0f789.456e123`" (or "`0f789.456e+123`") is the same as the one for the expression

```
operator 0f(
    operator 0I(
        operator 0I(
            operator 0I(#7),
            operator 0I(#8)
        ),
        operator 0I(#9)
    ),
    operator 0F(
        operator 0F(
            operator 0F(#4),
            operator 0F(#5)
        ),
        operator 0F(#6)
    ),
    operator 1E(
        operator 1E(
```

```
                operator 1E(#1),
                operator 1E(#2)
            ),
            operator 1E(#3)
)   )
```

The mandatory prefix "0f" for floating point numbers makes it easy for the scanner to already start constructing the correct syntax tree as soon as the first digit is found. If the fractional part ".456 is missing, ".0" is assumed, and, analogously, if the exponent part "e123 is missing, "e0" is assumed. For a single digit floating point literal "0f1", therefore the syntax tree for

```
operator 0f(
    operator 0I(#1),
    operator 0F(#0),
    operator 1E(#0)
)
```

is constructed by the scanner. A negative exponent, finally, corresponds to an additional application of "operator E-" to the exponent part, i.e., for "0f1e-12", the syntax tree for the following expression is produced:

```
operator 0f(
    operator 0I(#1),
    operator 0F(#0),
    operator E-(
        operator 1E(
            operator 1E(#1),
            operator 1E(#2)
)   )   )
```

Using floating point literals of some type Float is thus made possible by providing definitions for the following items:

```
val #0: Dig
  ...
val #9: Dig

fun operator 0I(d: Dig):         Int
fun operator 0I(i: Int, d: Dig): Int

fun operator 0F(d: Dig):         Frc
fun operator 0F(f: Frc, d: Dig): Frc

fun operator 1E(d: Dig):         Exp
fun operator 1E(e: Exp, d: Dig): Exp
fun operator E-(e: Exp):         Exp

fun operator 0f(i: Int, f: Frc, e: Exp): Float
```

Intermediate types Dig, Int, Frc and Exp can again be chosen as needed.

Part II

CODE GENERATION

Chapter 5

Code Generation Overview

The code generator for Mona modules gets the *annotated syntax tree* of the instantiated functor as input. This is produced by the type-checker when the *functor* is compiled. It consists of the syntax tree generated by the parser, with identifiers replaced by references to symbol table entries and type information added at some points required by the code generator. At instantiation time, i.e., when a Mona *module* unit describing an instance of the functor is compiled, the annotated tree is read from the central repository, together with the symbol table of the functor and the symbol tables of the modules supplied as its arguments. The *instantiator* links all symbols of the functor representing entities exported by module parameters to the corresponding symbols in the symbol tables of the actual argument modules. Thus all symbols in the annotated tree handed afterwards to the code generator contain complete implementation information.

The code generator produces definitions for all entities implemented in the instantiated functor in form of a C/C++ header file and an object file linkable by the standard UNIX linker `ld`. The object code is actually produced by the GNU C++ compiler from C++ code emitted by the Mona code generator. For the defined entities, external C names are chosen that uniquely identify these entities within the complete package hierarchy represented by the central repository. In addition to producing code defining these names, the code generator also stores them in the symbol table, augmenting the implementation information of the corresponding symbol.

The instantiator stores then (a simplified version of) the updated symbol table of the instantiated functor as module symbol table in the repository. The unique names of the exported entities suffice to generate code for usages of these entities when the module is supplied as argument to another functor. Thus it is not necessary to store the syntax tree with the module symbol table. This is only needed for functors to represent core language implementations of entities.

Mona allows the description of modules by nested functor applications or `let` blocks. As a result, usually code for more than one module is generated by a single run of the compiler for a `module` unit. There can even be cyclic dependencies between the modules produced by such a compiler run. This required splitting the described code generator functionality into several *phases*. Each of these phases is completed for all involved modules before the following phase is started.

A module for which code is available in the repository can be used either as argument for other functors to produce new modules, or externally from a C or C++ program. For the latter case, an *exporter* is provided that determines all object files in the repository required for using the module. Furthermore, it produces a C header file containing prototypes or preprocessor macro definitions for all items provided by the exported module.

In the following sections, annotated syntax trees are described, as well as the general structure of the code produced by the code generator. Furthermore, the different code generation phases are introduced. Finally, the exporter for Mona modules is sketched. A detailed description of the C++ code for all Mona constructs is given in the following chapter. This includes how expression evaluation is resolved in the code so that all temporary values are properly destroyed by appropriate destructor calls. A separate chapter discusses how the code generator exploits information provided by specifications, as well as several optimizations of the code, and the debugging information added to the code that allows to trace the execution of Mona sources.

5.1 Annotated Syntax Trees

As already mentioned, the annotated syntax trees used by the code generator are the syntax trees for the instantiated functor as produced by the parser, with identifiers replaced by symbol information. This means in detail that subtrees representing long identifiers $LONG\_ID$ as specified in Appendix C, as well as simple identifier nodes `id`, are replaced by special *symbol nodes*. These symbol nodes contain references to the symbol table entries representing the entity named by the identifier.

The symbols in the symbol table contain, besides their name and type, implementation information. This consists of the way they are defined and, depending on that, further information like, e.g., a unique key identifying a certain built-in value or operation for symbols defined as `builtin`.

In addition to identifiers being replaced by symbols, type expressions occurring in the tree, as e.g. in a sub-tree representing a `sizeof` expression, are replaced by special *type nodes*. A type node contains a reference to the entry of the type table representing the type expression replaced by the node.

The information available for the types in the type table contains the symbols representing the associated constructors and destructors. Also symbols representing conversion functions or special values fixed with one of the special type axioms like "`axiom type` *T* `truth` *f*" are attached to the type for which they are specified. Therefore the code generator needs to know the type of expressions at certain positions in the tree where such functions or values should be used. This type information is provided by type nodes added at such positions. These added type nodes will be mentioned in the descriptions of the code generated for the individual constructs that require them in Chapter 6.

5.2 Generated Code

The Mona code generator produces C++ code. Actually the code is C with a few extensions like `inline` definitions and reference parameter types provided by C++. The object code is generated from this by a call of g++, the GNU C++ compiler. In this section, we describe first the generated C++ source files and their interrelation. This includes a description of how inlining of function or procedure code is implemented. After that, we sketch the C++ code generated for the different constructs of Mona. A detailed description of the code is given in Chapter 6.

5.2.1 General Structure

As usual for C++, the C++ code consists of a *header* file and the actual *source* file. At the beginning of the source file, the header file is included with an `#include` preprocessor directive. The header file contains prototypes for all entities defined in the source file. Including it in another C++ source file allows thus to use these entities there.

Mona `inline` definitions are simply translated into C++ `inline` definitions. These have to be included in the *header* file, so that they are known and can be inlined when g++ is compiling a source file that uses the module, i.e., includes the module's header file. In fact, we keep inline definitions in a separate file and issue an `#include` directive for this file in the header file. Thus prototypes and inline definitions are not intermixed in the header file, but kept separate, though both are produced by a single tree traversal.

Before the *inline definitions file*, the header files of the argument modules supplied to the instantiated functor must be included, because entities exported by an argument module might already be used in one of the `inline` definitions. Finally, C++ `struct` and `union` definitions are written to the *structure definitions file* during code generation. This file has to be included before the list of prototypes, because the defined names for `struct` and `union` types are in general already used in prototypes. The header file generated for a module has thus the structure shown as Program 5.1 on page 58. It guarantees that all prototypes—for all involved modules—are known to g++ before any `inline` definition is processed, in particular also in case of cyclic module dependencies. The C++ type expressions used in the prototypes are independent from type definitions issued in header files of parameter modules. Therefore the order in which g++ finds the prototypes of a module and its parameters does not matter. With the above structure, the prototypes of the implemented module are processed by g++ *before* the prototypes of its parameter modules.

Per default, g++ produces static object code for `inline` functions, so that code is available in case inlining is not possible or the address of the `inline` function is needed somewhere. This would imply that copies of the object code for the `inline` functions provided by a module are contained in every object file using the module. We avoid this behavior by issuing certain g++-specific `#pragma`s at the beginning of the inline definitions file

```
#ifndef module-name_h
#define module-name_h

/**************************/
/* structure definitions */
/**************************/
#include "structure definitions file"

#ifdef __cplusplus
  extern "C" {
#endif

/*********************/
/* implemented items */
/*********************/
prototype
⋮

#ifdef __cplusplus
  }
#endif

/***********************/
/* parameter instances */
/***********************/
#include "parameter module header file"
⋮

/**********************/
/* inline definitions */
/**********************/
#include "inline definitions file"

#endif
```

Program 5.1: Structure of the Module Header File

```
#pragma interface "inline definitions file"

inline definition
⋮
```

and at the beginning of the actual source file, before the header file is included

```
#pragma implementation "inline definitions file"
#include "header file"

definition
⋮
```

This results in *external* code for the `inline` functions defined by a certain module *only* in the object file of this module. No object code is generated for `inline` definitions in included header files of parameter modules.

5.2.2 Code Sketch

The code generated for the different constructs of Mona is described in detail in Chapter 6. We sketch only the most important points here:

- Constructors and destructors are translated to C++ procedures. Constructor calls are inserted for local constant and variable definitions in command blocks, as well as for function result values given by a return command. Destructor calls are inserted for local constants and variables when their scope is left, as well as for all unnamed result values of expressions.
- Functions are translated to procedures constructing their result in the first parameter. Reference functions are functions that return *variables*, i.e., changeable values. In fact, the result is changeable only if the argument supplied for the `inref` parameter of the function is changeable. In the generated code, however, the result is always a reference, represented by a pointer to the function's result type.
- The commands for initializing and terminating module usage given with `init` and `term` definitions in the Mona source are collected in two parameterless procedures.
- Parameters are passed by reference. This is possible also for input parameters, since these are considered as *constants* in the defining body. The type checker prevents changes of formal input parameters. See, however, also the optimizations of parameter passing described in Section 7.2.
- Code for expressions consists of three parts:
  - code constructing the value of the expression in a temporary variable,

  - code representing the value of the expression in terms of the temporary variable, and
  - code for destroying the constructed value.

  In fact, both the construction and destruction part are split again into two parts that can be emitted separately. This is necessary for generating code that yields correct life-times for temporary values used as `inref` arguments for reference functions: The result of a nested reference function application will be a pointer to part of the innermost `inref` argument. This argument must hence live at least as long as the result pointer.

- Mona record and union types are translated to C++ `struct` and `union` types. These cannot be used unnamed as parameter types, since an unnamed `struct` or `union` is considered to be different from any other type in C++. Therefore a name is chosen and a definition is emitted to the structure definitions file if code for a not yet defined record or union type has to be emitted. Only the chosen name is then used as code.

- Every module uses its own names for record and union types. This requires arguments for entities provided by another module to be casted to the expected type. Therefore, preprocessor macros are provided for all procedures defined in the code for a module. They expand to calls of the corresponding procedures with the given arguments casted to the expected types. These macros get in fact the unique names chosen for the entities implemented by the procedures, not the procedures themselves.

The generated code is 'almost C'. No object-oriented features of C++ are used, only `inline` definitions and reference parameter types. We actually intend to replace reference types by pointers, thus generating C [KR88] code that can be compiled with the GNU C compiler gcc, which supports `inline` definitions.

5.3 Code Generation Phases

In Mona, modules are described as functor applications. These can, however, be arbitrarily nested. This requires the additional compilation of argument modules for which no code is available yet in the repository. Moreover, code might also have to be produced for modules resulting from functor applications occurring inside the instantiated functor. Thus code for more than one module might be produced by a single compiler run for a module unit.

Modules can also be described by `let` blocks. This allows convenient formulation of more complicated nesting structures. Moreover, this allows also the description of cyclic module dependencies through the use of `forward` declarations for local modules (cf. the definition of the syntax domain *module* in Appendix A). Again, cyclic dependencies can also result from recursive module definitions produced with functor applications and `forward` declared modules inside the instantiated functor. Such cyclic dependencies should be avoided, but since this is in some cases very hard, they are supported in Mona.

Emitting code for a module is only possible if the implementation information of the symbols provided by parameter modules is already augmented with the external names chosen for these symbols. In the case of cyclic module dependencies, it is not possible to choose an appropriate order of generating code for the modules involved in the cycle. Therefore choosing the external names and augmenting the corresponding symbol information is done in a separate *mark phase* for all modules produced in a certain compiler run, before emitting code for one of them is started.

Furthermore, generating object code for C++ source files with a C++ compiler will only work out if all parameter header files included directly or indirectly are already emitted. Again this cannot be achieved in case of cyclic module dependencies if the C++ compiler is invoked immediately after the C++ code is emitted for each module. Therefore the C++ compiler is only invoked in a separate *compile phase*, after the C++ source files for all involved modules are already emitted.

5.3.1 Marking

During the mark phase for a single module, the annotated syntax tree for the instantiated functor is traversed down to the level of definitions. For each definition found, except for `forward` definitions, an external name is chosen, and the implementation information of the defined symbol is augmented correspondingly. Module definitions, import directives and functor usage declarations are ignored, because no code has to be produced for them. Also `init` and `term` definitions are ignored, because no symbols are associated with them.

5.3.2 Emitting

In the emit phase, the annotated tree is traversed once more. This time C++ code as described in detail in Chapter 6 is emitted for all definitions found. Again, module definitions, import directives and functor usage declarations are ignored. Also `forward` definitions can be ignored: the `#include` structure of the files as described in the previous section guarantees that every usage of a name is preceded by a corresponding prototype or preprocessor macro definition in the expanded source file.

References to the trees representing the commands from `init` and `term` definitions found during the tree traversal are collected in two lists. From these lists, the definitions of the initialization and termination procedures for the module are generated after the tree has been traversed completely.

5.3.3 Compiling

The compile phase for a module consists just of a call of g++ for the C++ source file generated in the emit phase. After successful execution of g++, the C++ source file can be removed, so that only the components of the header file and the object file remain in the repository.

5.4 Exporting Modules

The header and object files produced by the code generator could in principle also be used from external C or C++ programs. However, this would require exact knowledge of how the module was built, since the object code for the module refers to object files of parameters, which in turn refer to their parameters, and so on. Thus all these object files must be linked to the external program. Moreover, also the initialization and termination procedures provided by all involved modules have to be called in correct order in the external program.

To ease the use of Mona modules in external C programs, and also to hide the internal structure of the central repository, an *exporter* is provided. It generates the list of all object files in the repository required for using a requested module in form of a macro definition that can be included in a `makefile` for use with the UNIX `make` utility. Furthermore, it generates a C header file that contains an `#include` directive for the module header file in the repository, as well as definitions of an initialization and a termination macro as appropriately ordered sequences of calls of the initialization, or termination, respectively, procedures of all involved modules. Finally, it adds also preprocessor definitions to the generated header file that make the entities provided by the exported module available with names that resemble Mona (long) identifiers as close as possible. Thus the user is relieved from having to determine the exact dependencies of the requested module, as well as from having to know the locations of the involved code files in the repository and the unique names identifying the entities provided by these files.

The list of object files, as well as the sequence of initialization and termination procedure calls, is generated from a *module dependency graph* derived from dependency information stored in the repository. Each node in this graph represents a module. The information stored for each module consists of the unique module name as described in Section 6.6, the location of the code files for the module in the repository, and links to all nodes representing parameter modules. This graph is traversed, starting at the module to export, to produce a list of all contained modules in depth-first order. Every visited module in the graph is marked before its first parameter module is visited, so that duplicate entries in the produced module list can be avoided and cycles are detected. The order of the modules in the produced list represents an admissible initialization order, and termination can be done in reverse order. For the list of object files, order is unimportant.

The preprocessor definitions in the header file are produced from the symbol table of the module to export. For each symbol in the table, a C identifier resembling the Mona name of the symbol is defined as the unique external name chosen by the code generator for the symbol. For type symbols, additional definitions are generated for all constructor and destructor symbols associated with the type represented by the symbol. Symbols representing modules do not contain an external name assigned by the code generator. Their implementation information consists of a reference to the symbol table of the represented module. Definitions are generated for all symbols in this symbol table instead of for the module symbol itself, with the name of the module symbol prepended to all defined identifiers. All identifiers defined in the export header file get per default the name of the exported module as prefix. However, a different prefix, or no prefix at all, can be chosen.

The following has to be noted when using an exported module:

- The initialization macro has to be executed before any other entity is used.
- Functions and constructors are available as procedures that construct values in their first argument. When calling such a procedure, the first argument therefore must not have been constructed (or must have been destroyed) before.
- Destructors are available as procedures that destroy their first argument.
- Variables declared with a type provided by the used module must be constructed by calling a procedure representing a suitable function or constructor before they can be used elsewhere. They must be destroyed by calling a procedure representing a suitable destructor before their scope is left.
- The termination macro has to be executed before program execution terminates.

See Appendix E for an example of an export interface and its usage.

Chapter 6

The Code in Detail

This chapter contains a detailed description of the C++ code generated for the different constructs of Mona. We intend to switch to producing C code instead. Some preparations for this are already done. Declarations of local variables, e.g., occur only at the beginning of blocks, since C, in contrast to C++, does not allow to intermix declarations and commands. The only important C++ features used in the code, which are not available in C, are `inline` definitions and reference types in parameter declarations. Getting rid of the latter one involves some changes of the generated procedure definitions and the code generated for expressions. Parameters with pointer types have to be used instead. Thus, addresses of actual parameters have to be taken explicitly. This can be done in the type-casting macros anyways provided for defined procedures—see Section 6.5 below. Furthermore, formal parameters have to be dereferenced explicitly in the bodies of generated procedure definitions. Implementing the inlining of function or procedure bodies in the Mona code generator requires more effort. However, the GNU C compiler `gcc` supports `inline` definitions as a C language extension. This will be used in a first step towards C code.

Actually, the generated code might look a bit different from what is described here, since some additional optimizations are applied. These are described in Chapter 7. The complete code generated for a simple example module can be seen in Appendix E.

6.1 Definitions

In this section, we will first specify the code generated for the Mona items implemented in the core language Cora. The code for each item consists usually of a declaration (a *prototype*) written to the module *header* file, and a definition written to the *source* file. For generated procedure definitions, the prototype to be written to the header file is just the head of the definition; we will not mention this separately in every case. Moreover, an additional macro definition for casting arguments to the types expected by the procedure is generated in this case. This macro gets actually the name of the procedure, and the procedure's name gets an underscore prepended. This is discussed in Section 6.5. For Mona items tagged as `inline`, also the C++ prototype and definition are tagged with `inline`, and the definition is written to the *inline definitions* file instead of to the source file.

Afterwards we will treat `alias`, `builtin` and `extern` definitions. For entities declared with `forward`, definitions have to be given later on. Only for these definitions code is generated. Mona *functions* can also be defined with `default`. This means that a default implementation of the function has to be generated from the function's *specification*. A section of Chapter 7 is dedicated to this issue.

Only the entities defined in a module that are listed in the specification named in the head of the `module` unit can be used in other modules or external programs. However, an exported function or procedure defined as `inline` may well use entities visible only locally in the module. The code of such an entity must then be visible also where the function or procedure is inlined. Therefore we make the code for all items defined in a module (also items defined locally in `let` blocks in the module) externally visible.

No code is produced for module definitions. A module can only be defined as one of the parameter modules or as application of a functor introduced by a *functor usage declaration* (or as module exported by one of these). In each case, code for this module is available separately in the central repository.

Variables are used in the following descriptions to denote certain Mona constructs. The same variables with additional superscript "^{C}" are used to denote the C++ code generated for these constructs. The variables are:

param denotes a Mona parameter declaration—the C++ code *param*^{C} generated for parameter declarations is described in Section 6.2;

cmd denotes a Mona command—the C++ code *cmd*^{C} generated for commands is described in Section 6.3;

type denotes a Mona type expression, represented by a type table entry in the annotated syntax tree—the C++ code *type*^{C} generated for types is described in Section 6.5;

id denotes a Mona identifier, represented by a symbol in the annotated syntax tree—the C++ code, i.e., identifier, *id*^{C} generated for symbols is described in Section 6.6.

6.1.1 Types

For Mona type definitions, C++ `typedef`s are generated in the header file. However, these definitions are not used by the code generator. They are only provided for external usage of the module.

For a Mona type definition of the form

```
type id = type
```

the following code is written to the header file:

```
typedef typeC idC;
```

A type defined with `new` is implemented as its representation type. This means that the same `typedef` as above is also generated for

> `type` *id* `new` *type*

6.1.2 Constructors and Destructors

The major problem we faced in generating C++ code for Mona was that Mona supports constructors and destructors for its types. In particular, code for implicit destructor calls destroying intermediate results of nested expressions has to be generated. Without this, a more or less one-to-one translation from Mona core language definitions to C++ would be possible.

Mona uses constructors and destructors in a similar manner as C++ does. However, C++ supports the definition of constructors or destructors only for *classes*, and not for atomic types like `int`. Translating Mona constructors and destructors to C++ constructors and destructors would therefore require translating all Mona types to C++ classes. For efficiency reasons, we decided not to do this. Instead, we translate constructors and destructors to ordinary procedures (i.e., C++ functions with result type "`void`") and produce code for their implicit calls ourselves.

For a Mona constructor definition of the form

> `constr` id_1`:` *type*`'`id_2`(`*param*`, ...)` *cmd*

the following C++ procedure definition is generated:

> `void` id_2^{C}`(`$type^{C}$ `&`id_1^{C}`,` $param^{C}$`, ...) {` cmd^{C} `}`

The identifier id_1 names the constructed value. It is added as reference parameter to the parameter list of the C++ procedure. The constructor identifier id_2 is optional in the concrete syntax. In the annotated syntax tree, however, the symbol representing the constructor is always available at the corresponding place.

Code of the same form is produced for a Mona destructor definition

> `destr` id_1`:` *type*`'`id_2`(`*param*`, ...)` *cmd*

Only *default destructors*, i.e., unnamed, parameterless destructors specified like

> `destr` *id*`:` *type*`()`

are called implicitly when the scope of variables is left. All other forms can only be called explicitly, e.g. in bodies of other destructor definitions.

6.1.3 Procedures

A Mona procedure definition of the form

proc *id*(*param*, ...) *cmd*

is translated to the C++ procedure definition

void id^{C}($param^{\mathsf{C}}$, ...) { cmd^{C} }

6.1.4 Functions

For being able to call destructors for all intermediate results in nested expressions, it is necessary that these intermediate results are explicitly assigned to temporary variables declared for this purpose. A call of a function `f` with result type `T` in the code for a Mona expression can thus only have the form

```
T tmp; tmp = f(...);
```

so that afterwards the procedure representing the appropriate destructor can be called for the temporary variable `tmp`. To make this point clear also for the external usage of Mona code, which requires calling destructor-procedures by hand, we translate Mona functions to C++ procedures with the result constructed in the first argument. The above function call takes thus actually the form

```
T tmp; f(tmp, ...);
```

With this, it is not possible to call a function without being afterwards able to destroy the result value. Still, the C programmer using a Mona module is of course responsible for actually calling the destructor.

A Mona function definition of the form

fun *id*(*param*, ...): *type cmd*

is thus translated to the C++ procedure definition

void id^{C}($type^{\mathsf{C}}$ &_, $param^{\mathsf{C}}$, ...) { cmd^{C} }

A single underscore "_" is used as identifier for the result parameter—it is referred to in the code generated for the `return` command specifying a return value.

A function can also be defined with an explicitly named result value in Mona:

fun id_1(*param*, ...) id_2: *type cmd*

In this case, *cmd* is an ordinary procedure body, where `return` commands are not allowed to have an expression attached. The named return value has to be constructed by an explicit constructor call in the body. The generated code for this kind of function definition is:

void id_1^{C}($type^{\mathsf{C}}$ &id_2^{C}, $param^{\mathsf{C}}$, ...) { cmd^{C} }

6.1.5 Reference Functions

Mona *reference functions* are functions that return *variables*. This means that it is possible to assign to an application of a reference function, or to use it as output or input/output parameter in any other procedure call. In C++, functions with a reference type as result type show this behavior.

Reference functions serve for allowing updates of data structures via selectors. Consider, e.g., a module encapsulating some data structure implementing an array of elements of some type `Elem`. It could provide a reference function

```
reffun element(i: Index): Elem
```

for getting the value of the `i`th element of the array, but also to update this element.

Also, if the module does not only encapsulate an array data structure, but provides arrays as an abstract data type `Array`, one could provide an access function

```
reffun operator [](a: Array, i: Index): Elem
```

to allow destructive access to elements of an array a with the usual notation `a[i]`. However, in this case a slightly different behavior is desirable: if the array a is a *constant*, then also `a[i]` should be constant, so that it is not possible to change a by changing its elements. For this purpose, an additional parameter mode "`inref`" was introduced in Mona for use with reference functions. The described behavior of the array indexing operator is achieved if the above specification of "`operator []`" is replaced by

```
reffun operator [](inref a: Array, i: Index): Elem
```

In this case, the result of `a[i]` can be changed only if also a itself can be changed. Only reference functions can have `inref` parameters, and at most one parameter of a reference function can be tagged with `inref`.

Translation of a reference function to a procedure would require a result parameter of a "reference to a reference" type. Since this is not possible in C++, we use a reference to a pointer instead. Thus we generate for a Mona reference function definition of the form

reffun *id*(*param*, ...): *type cmd*

a C++ procedure definition of the form:

void id^C($type^C$ *&_, $param^C$, ...) { cmd^C }

This requires to assign the *address* of the return value to the result parameter "_" inside the function, and to explicitly *dereference* the result value for using it.

6.1.6 Constants and Variables

For a global Mona *variable* definition of the form

```
var id: type
```

a C++ `extern` declaration

```
extern type^C id^C;
```

is written to the module header file, and a definition

```
type^C id^C;
```

is written to the source file.

Constants defined globally in a Mona module are in fact only constant for *users* of this module. *Inside* the module they are treated as variables, so that they can, e.g., be initialized with constructor and procedure calls in several steps in `init` definitions (see below). This is resolved by generating a C++ variable definition as above in the source file for a Mona constant definition of the form

```
val id: type
```

and generating an `extern` declaration in the header file of the form:

```
extern type^C val id^C;
```

The keyword "`val`" occurring in this declaration is defined at the beginning of the list of prototypes in the header file as

```
#ifdef module-name_cc
  #define val
#else
  #define val const
#endif
```

and is undefined with "`#undef val`" at the end of the prototype list. Thus, it will be replaced by the preprocessor with `const` if the header file is included in some source file *using* the module. In the source file *implementing* the module, however, "*module-name*`_cc`" is defined with

```
#define module-name_cc
```

before the header file is included. Thus, "`val`" is replaced by the empty string in this case, resulting in a `variable` declaration for the Mona constant in the header file. Note that `val` is a Mona keyword and thus cannot collide with any identifier occurring in the prototype list.

The code generated for *local* constant and variable definitions is described together with the code for the command blocks in which they occur in Section 6.3.

6.1.7 Module Initialization and Termination

A Mona module is initialized by executing all commands given by `init` definitions in textual order. Termination is done by executing all commands given by `term` definitions in reverse textual order. The code generator produces two parameterless procedures for this purpose. Their prototypes in the header file are:

```
void _init_module-name();
void _term_module-name();
```

with *module-name* being the unique identifier for the module described in Section 6.6.

To avoid multiple initialization and termination of a module, a 'reference count' is kept in the source file:

```
static int initialized = 0;
void _init_module-name() {
  if (!initialized++) {
    ⋮
  }
}
void _term_module-name() {
  if (!--initialized) {
    ⋮
  }
}
```

The body of the initialization procedure will only be executed if `initialized` has a value of zero. Further calls of the procedure only increase the value of `initialized`. With each call of the termination procedure, the value will then be decreased, until the body of the procedure is executed when the value zero is reached.

6.1.8 Aliases

For the definition of some Mona entity id_1 as an *alias* for another entity id_2, a preprocessor macro definition like the following is written to the header file:

```
#define id1^C id2^C
```

Mona allows to specify *representation casts* for selected arguments of a constructor, destructor, procedure or function defined as alias. This allows to provide functionality for a `new` type as aliases for corresponding functionality already implemented for the representation type. Consider, e.g., a type `P` implemented as a two-component record:

```
type P new Record[
           fst: F
           snd: S
         ]
```

For the record type, the component selection operators "`.fst`" and "`.snd`" are already available as built-in reference functions. They can be provided also for the `new` type with:

```
reffun operator .fst(inref p: P): F alias operator .fst(repr)
reffun operator .snd(inref p: P): S alias operator .snd(repr)
```

Since the C++ representation for a `new` type is the same as for its representation type, no code has to be produced for the representation casts.

6.1.9 Built-in Constructs

Also for `builtin` definitions, code is only produced in the header file. For a `builtin` type definition

`type` *`id`* `builtin` *`builtin-type`*

a `typedef` of the following form is generated:

`typedef` *`builtin-type`*^{C} *`id`*^{C}`;`

For a `builtin` constant definition

`val` *`id`*`:` *`type`* `builtin` *`builtin-val`*

a `static const` definition as follows is generated:

`static` *`type`*^{C} `const` *`id`*^{C} `=` *`builtin-val`*^{C}`;`

For all other kinds of `builtin` definitions, preprocessor macro definitions are produced. Appendix D lists all built-in constructs available in Mona, together with the code that is produced for each of them.

6.1.10 Externally Defined Items

For entities defined as "`extern` *`extern-id`*", a declaration for *`extern-id`*, as well as a preprocessor macro definition making the externally defined item accessible via its Mona name, is written to the header file. An externally defined function also gets a procedure-interface through the macro definition. For a function definition like

`fun` *`id`*`(`id_1`:` $type_1$`, ...):` *`type`* `extern` *`extern-id`* a prototype and macro definition of the following form are generated:

$$\mathit{type}^{\mathsf{C}}\ \mathit{extern\text{-}id}(\mathit{type}_1^{\mathsf{C}}\ \mathit{id}_1^{\mathsf{C}},\ \ldots);$$
$$\texttt{\#define}\ \mathit{id}^{\mathsf{C}}(\_,\ \mathit{id}_1^{\mathsf{C}},\ \ldots)\ ((\_) = \mathit{extern\text{-}id}(\mathit{id}_1^{\mathsf{C}},\ \ldots))$$

All parameters of an external function or procedure are per default assumed to have to be passed by value (like usual in C), independent of the given parameter modes. The parameter types in the prototype for the external function are thus exactly the C++ equivalents of the parameter types given in the Mona definition. Call by reference can be explicitly specified for selected arguments, using, e.g., "`extern` *extern-id*`(-, &, -)`" to state that the second argument must be a *reference* to a value of the type given in the parameter list. An "`&`" is then appended to the corresponding parameter type in the prototype, or a "`const &`" in case of an `in` or `inref` parameter.

6.2 Parameter Declarations

All parameters are passed by reference in Mona (except for externally defined items—see above). For input parameters this is possible because they are considered to be *constants* within the body of the definition. The Mona type checker guarantees that they are not changed. Still, some restrictions apply: aliasing via parameters is not allowed, i.e., no `in` parameter may occur as (part of an) `inout` or `out` parameter (and vice versa) in the same procedure call.

Mona parameter declarations thus map to C++ parameter declarations as follows:

$$\begin{array}{lll}
\texttt{in}\ \ \ \ \ \mathit{id}\texttt{:}\ \mathit{type} & \longrightarrow & \mathit{type}^{\mathsf{C}}\ \texttt{const}\ \&\mathit{id}^{\mathsf{C}} \\
\texttt{inref}\ \mathit{id}\texttt{:}\ \mathit{type} & \longrightarrow & \mathit{type}^{\mathsf{C}}\ \texttt{const}\ \&\mathit{id}^{\mathsf{C}} \\
\texttt{inout}\ \mathit{id}\texttt{:}\ \mathit{type} & \longrightarrow & \mathit{type}^{\mathsf{C}}\ \&\mathit{id}^{\mathsf{C}} \\
\texttt{out}\ \ \ \mathit{id}\texttt{:}\ \mathit{type} & \longrightarrow & \mathit{type}^{\mathsf{C}}\ \&\mathit{id}^{\mathsf{C}}
\end{array}$$

Actually, we optimized this to achieve that values that might reside in registers are not forced into storage if used as input parameters. This is discussed in Section 7.2.

6.3 Commands

In this section, the C++ code generated for the different kinds of Mona commands is described.

6.3.1 Block

A Mona command block of the form

```
{
  localDef | cmd
  ⋮
}
```

is translated to the following C++ code:

```
{
  localDef^C | cmd^C
  ⋮
  destroy local values
}
```

To be able to generate code for destroying local values, the symbols representing these values are stored in a stack during code generation. The corresponding destructors can be determined via the types of these symbols. When generating code for a block, the stack depth at the beginning of the block is remembered. At the end of the block, symbols are popped from the stack, and the appropriate destructor calls are generated, until the remembered stack depth is reached. The destruction of the local variables takes thus the form of a sequence of C++ procedure calls

```
dd^C(id^C); } ...
```

where *dd* is the symbol representing the default destructor for the type of the symbol *id* from the local value stack. The code id^{C} generated for a symbol is a C identifier. The chosen identifiers are described in Section 6.6. The right brace "}" closes the block opened at the code for the definition of the local value *id*—see below.

Symbols are pushed to the stack when code for local definitions *localDef* is generated. The C++ code for a local *variable* definition

```
var id_1: type'id_2(expr, ...)
```

consists of a variable declaration and code for a constructor call:

```
{ type^C id_1^C;
  constr(expr), ...
  id_2^C(id_1^C, value(expr), ...);
  ..., destr(expr)
```

The block opened for the declaration is closed with the code for the corresponding default destructor call as shown above. The constructor identifier id_2 is optional in the concrete syntax. In the annotated syntax tree, however, the symbol representing the constructor to call is always available at the corresponding place.

Code for expressions *expr* consists always of three parts, as shown for the constructor arguments above:

$constr($**expr**$)$ is code for constructing the value of the expression in a temporary variable declared for this purpose,

$value($**expr**$)$ is code representing the value of the expression in terms of the temporary variable, and

$destr($**expr**$)$ is code destroying the value of the temporary variable holding the result of the expression.

Arguments for a procedure call in the code are destroyed in the reverse order of their construction. Section 6.4 describes the code actually generated for the different kinds of Mona expressions.

Local *constants* are implemented as local C++ variables as well, so that it is possible to do the construction of the constant's value with a constructor call independent from the declaration. Therefore the same code as given above for local variables is also generated for a local constant definition

```
val id1: type‘id2(expr, ...)
```

6.3.2 If and If-Else

For a Mona `if` command of the form

```
if (expr) cmd1 else cmd2
```

a C++ `if` command of the following form is generated:

```
if (({int __;
       constr(t(expr))
       __ = value(t(expr));
       destr(t(expr))
       __;})) cmd1^C else cmd2^C
```

It uses a *command expression* "`({`cmd`; ...})`" for evaluating the value of the `if` condition in a variable "`__`" local to this expression. Command expressions are a C language extension provided by the GNU compiler.

The actual condition evaluated is "`t(expr)`". The *truth* function `t` for the type *T* of the condition "*expr*" issued in the Mona source is additionally applied. This function maps the result of *expr* to the built-in `Bool` type of Mona, which is implemented as the C++ integer type `int`. It is specified with an axiom of the form "`axiom type T truth t`" and can be obtained by the code generator via the type table entry for *T*. This type table entry is available as a type node for the type of *expr* added to the syntax tree of the `if` command by the type checker.

For a Mona `if` command without `else` branch, the `else` branch of the `if` command in the above C++ code is simply omitted.

6.3.3 Switch

The Mona `switch` command is translated to a cascading `if` command. Translation to a C++ `switch` is not possible in general, because Mona does not require that the expressions describing the individual cases can be evaluated at compile-time.

For a Mona `switch` command of the form

```
switch (expr) opt
```

a C++ dummy loop is generated:

```
do {
  int __;
  constr(c(expr))
  __ = value(c(expr));
  destr(c(expr))

  opt^C
} while (0);
```

This loop contains the `if` cascade *opt*^{C}. For a list of cases

```
case expr: case ...: cmd
⋮
```

it takes the form:

```
if (__ == ({int __;
            constr(c(expr))
            __ = value(c(expr));
            destr(c(expr))
            __;}) ||
    __ == ...
    ) cmd^C
else
⋮
```

A default clause

```
default: cmd
```

is translated to

```
if (1) cmd^C
```

The function c applied to the `switch` expressions is the *case* function specified with "`axiom type` T `case` c" for the type T of the expression *expr*. It maps the value of the expression to the built-in `Int` type of Mona, which is represented in the code as C++ `int`. The code generator obtains the case function via a type node for the type of *expr* added to the syntax tree of the `switch` command. Separate type nodes are contained for each case expression, so that these actually need not be of the same Mona type.

The enclosing `do` loop serves for the correct effect of C++ `break` commands generated for Mona `break` commands occurring in the case list: *there is* a loop that can be left with a `break`, and *only this* loop is left, not an enclosing one.

To enable generating the correct destructor calls for a `break` command, the *break depth* is maintained in a global variable during code generation. This is the depth of the local value stack down to which destructors should be called in case of a `break`. Before emitting code for a `switch` command, the current break depth is saved and then set to the current stack depth. Thus only local values defined inside the `switch` will be destroyed by a `break` inside the `switch`. When the `switch` code is emitted, the original break depth is restored.

6.3.4 While and Do-While

For a Mona `while` loop of the form

```
while (expr) cmd
```

C++ code analogous to the code for an `if` command is generated:

```
while (({int __;
          constr(t(expr))
          __ = value(t(expr));
          destr(t(expr))
          __;})) cmd^C
```

The truth function t is again obtained via an added type node representing the type of the condition.

Additionally, the break depth is set and reset like during the code generation for a `switch` command. In the same way, the *continue depth* is handled. It is used to determine which destructors to call in case of a `continue` command. The continue depth has to be maintained separately since a `continue` inside a `switch` command refers to the enclosing *loop*, in contrast to a `break` at the same place, which refers to the `switch`.

A Mona `do` loop

```
do cmd while (expr)
```

is handled in the same way. C++ code of the following form is generated:

```
do {
  cmd^C
} while (({int __;
           constr(t(expr))
           __ = value(t(expr));
           destr(t(expr))
           __;}));
```

6.3.5 Break and Continue

A Mona `break` or `continue` is implemented by the corresponding C++ command. However, additional destructor calls for local variables have to be generated. The code for a `break` command is thus of the form

```
{
  destroy values on local value stack
    down to break depth
  break;
}
```

and the code for a `continue` command is of the form

```
{
  destroy values on local value stack
    down to continue depth
  continue;
}
```

Though destructors are called for symbols on the local value stack, these symbols are of course *not* popped off the stack. This is only done when code for the *static* end of a scope is generated.

6.3.6 Return

A `return` command *without* result expression in a Mona procedure body is translated to a sequence of destructor calls for local values and a C++ `return` command. Since procedures, functions, constructors or destructors cannot be local to command blocks in Mona, *all* values on the local value stack have to be destroyed in case of a `return`. Thus the following code is generated:

```
{
  destroy all values on local value stack
  return;
}
```

Again, the destroyed values are *not* popped off the stack.

For a `return` *with* result expression

```
return expr
```

occurring in the body of Mona *function* definition (without named result value), additionally code for constructing the result parameter "_" with the value given by *expr* has to be generated. This has of course to be done before local variables are destroyed. Thus the following C++ code is generated:

```
{
  constr(expr)
  cc^C(_, value(expr));
  destr(expr)
  destroy all values on local value stack
  return;
}
```

The symbol *cc* representing the copy constructor to be called for constructing the result parameter "_" is obtained from the type table entry representing the function's result type. This is stored in a global variable before code for the body of a function definition is emitted.

Finally, if the `return` occurs in the body of a *reference function* with result type *T*, the following code is generated:

```
{
  constr(expr)
  _ = &value(expr);
  destr(expr)
  destroy all values on local value stack
  return;
}
```

The *address* of the result value is assigned to the result parameter "_"; no constructor is called.

6.3.7 Procedure Call

The C++ code for a Mona procedure call

```
id(expr, ...)
```

is of the form

```
{
  constr(expr), ...
  id^C(value(expr), ...);
  ..., destr(expr)
}
```

The actual parameters are evaluated, the procedure is called with their values, and afterwards the parameter values are destroyed in reverse order of their construction.

6.3.8 Constructor and Destructor Call

Code for a Mona *constructor* call of the form

```
constr expr₁: type'id(expr₂, ...)
```

is analogous to the code for a procedure call, given that the constructor is implemented as procedure constructing its first parameter:

$$
\begin{array}{l}
\texttt{\{} \\
\quad constr(\texttt{expr}_1),\ constr(\texttt{expr}_2),\ \ldots \\
\quad \texttt{id}^{\mathsf{C}}\texttt{(}value(\texttt{expr}_1)\texttt{, } value(\texttt{expr}_2)\texttt{, } \ldots\texttt{);} \\
\quad \ldots,\ destr(\texttt{expr}_2),\ destr(\texttt{expr}_1) \\
\texttt{\}}
\end{array}
$$

The called constructor can be unnamed in the concrete syntax, i.e., the identifier *id* might not be given. In the annotated syntax tree, however, the symbol representing the called constructor is always available at the corresponding place.

Exactly the same code is generated for a *destructor* call of the form

```
destr expr₁: type'id(expr₂, ...)
```

6.4 Expressions

As already mentioned, code for expressions *expr* is always emitted in three parts: *constructing* the value of the expression in a temporary variable ($constr(\texttt{expr})$), using the *value* of the temporary variable ($value(\texttt{expr})$), and *destroying* the temporary variable ($destr(\texttt{expr})$). The scope of temporary variables is explicitly delimited with braces. Before the declaration of a temporary variable for the result of an expression in $constr(\texttt{expr})$ a left brace "{" is emitted, and after the code for destroying the value of the temporary variable in $destr(\texttt{expr})$ the corresponding right brace "}" is written. This requires that code for destroying temporary values is emitted in reverse order of their construction.

Furthermore, reference functions with `inref` parameters require special treatment. Reference functions do not produce values themselves, but return only *references* to some data constructed and destroyed elsewhere. These references are represented as pointers, i.e., the temporary variable for the result of a reference function is a pointer to the result type of the function. If the reference function has an `inref` parameter, its return value will usually be a reference to (part of) the argument supplied for this parameter. The life-time of this parameter thus must not end before the life-time of the result of the reference function. In the code for a reference function, the `inref` argument must be constructed *before* the declaration of the temporary variable for the result pointer, and may be destroyed only after the end of the scope of this result pointer. Moreover, if the actual `inref` parameter is again the application of a reference function, it is the `inref` argument of *this* function which has to be constructed first and may only be destroyed after the scope of the reference function results has ended. Consider the following:

```
val v: T

fun f(a: T): U

reffun r(inref a: U): V
reffun s(inref a: V): W

proc p(in a: W)
```

The following procedure call might be issued with these items:

```
p(s(r(f(v))))
```

In the code for this procedure call, the temporary variable tmp_f for the result of f(v) must be declared first and destroyed last, since the temporary variable tmp_s for the result of s(r(f(v)) is a pointer that will usually point to some part of the value of tmp_f:

```
{ T^C tmp_f;
  f^C(tmp_f, v^C);
  { W^C *tmp_s;
    { V^C *tmp_r;
      r^C(tmp_r, tmp_f);
      s^C(tmp_s, tmp_r);
    }
    p^C(tmp_s);
  }
  dd_T^C(tmp_f)
}
```

The code for constructing the value of the argument f(v) of r in tmp_f is thus separated from the code constructing the result of r. It must be already emitted as first part of the code $constr$(s(r(f(v)))) constructing the result of s. Let us call this part $constrInref$(r(f(v))): it constructs the value of the inref argument of s that must live at least as long as the result pointer of s. It is, in fact, the same as the code $constrInref$(f(a)) that had to be emitted as first part of $constr$(r(f(a))) if r(f(a)) was not used as inref argument. And this, in turn, is the code that would be generated as $constr$(f(a)) if f(a) was not used as inref argument. The code $constr$(*expr*) generated for constructing the value of an expression *expr* depends thus in general on the mode of the parameter for which *expr* is supplied.

Analogously, the code for destroying the value of f(a) stored in tmp_f with the default destructor dd_T for type T must only be emitted as last part of the code $destr$(s(r(f(v)))) destroying the result of s. We call this code $destrInref$(r(f(v))). It is the same as the code $destrInref$(f(v)) that had to be emitted as last part of $destr$(r(f(a))) if r(f(a)) was not used as inref argument, and also the same as what would be generated as $destr$(f(a)) if f(a) was not used as inref argument. Thus, also the code $destr$(*expr*) for destroying the value of an expression *expr* depends in general on the mode of the parameter for which *expr* was supplied.

In the following subsections, the generated C++ code for the different parts of expression code is described for all kinds of Mona expressions, with the *constrInref* and *destrInref* parts separated. Parts of the code that are only emitted if the value of the expression is not used as `inref` parameter are marked with "*if* `!inref`:".

The temporary variable names *tmp* used in the code are of the form "_*count*", where count is the value of the global *temporary variable counter* maintained during code generation. Its initial value is "0", so the first temporary variable name used is "_0'. When a declaration of a temporary variable is emitted, the counter is incremented, and when the corresponding closing brace ending the scope of the variable is emitted, the counter is decremented.

6.4.1 Function Application

For a function application of the form

> `id(expr, ...)`

the following code is generated, assuming that the result type of the function `id` is *T*:

> *constrInref*:
>> `{` T^{C} *tmp*`;`
>> *constr*(`expr`), ...
>> $\mathtt{id}^{\mathsf{C}}$`(`*tmp*`,` *value*(`expr`)`, ...);`
>> ..., *destr*(`expr`)
>
> *constr*:
>> *if* `!inref`: *constrInref*(`id(expr, ...)`)
>
> *value*:
>> *tmp*
>
> *destr*:
>> *if* `!inref`: *destrInref*(`id(expr, ...)`)
>
> *destrInref*:
>> $\mathtt{dd}^{\mathsf{C}}$`(`*tmp*`);`
>> `}`

The *constrInref* code will always be emitted first. It declares a temporary variable *tmp* for the result. Then the arguments are constructed and the procedure implementing the function is called with the values of the arguments. It constructs the result of the function in the temporary variable given as first parameter. Afterwards the arguments can immediately be destroyed in the reverse order of their construction.

The *constr* code is just the *constrInref* code, except if the result is used as `inref` argument for some reference function `r`. In this case, the *constrInref* code was already emitted before the temporary result pointer for the application of `r` was declared. Thus no more code has to be emitted.

The *value* of the application is available in the temporary variable *tmp* after construction.

The *destr* code is just the *destrInref* code, except if the application was used as `inref` argument for a reference function `r`. In this case, the *destrInref* code will only be emitted after the end of the scope of the result pointer for the application of `r`.

The *destrInref* code finally destroys the result value of the function application stored in the temporary variable with a call of the procedure representing the default destructor *dd* for the result type *T* of the applied function.

If the applied function `id` is in fact a *reference* function, the code is different. Assuming that the first parameter of `id` is the `inref` parameter, it looks like follows:

constrInref:

  constrInref(`expr`)

constr:

  if `!inref`: *constrInref*(`id(expr, ...)`)
  `{` T^{C} `*`*tmp*`;`
    constr(`expr`), ...
    $\mathtt{id}^{\mathsf{C}}$`(`*tmp*`,` *value*(`expr`)`, ...);`
    ..., *destr*(`expr`)

value:

    `*`*tmp*

destr:

  `}`
  if `!inref`: *destrInref*(`id(expr, ...)`)

destrInref:

  destrInref(`expr`)

The *constrInref* code for constructing the value of the `inref` argument of some enclosing reference function application is just the code *constrInref*(`expr`) for constructing the `inref` argument `expr` supplied to `id`. If `id` has no `inref` parameter, no code is produced for this.

The *constr* code starts with the *constrInref* code, except if this was already emitted before, i.e., if the reference function application is used as `inref` argument. Afterwards the temporary result pointer *tmp* for the application is declared and initialized by constructing the arguments and calling the procedure representing `id` with *tmp* as first parameter. Finally, the arguments of `id` are destroyed. Note, however, that this will not destroy the value of the `inref` argument `expr`. The *destr*(`expr`) code does not call a destructor if `expr` is used as `inref` parameter—cf., e.g., the function application code above. The value of `expr` is only destroyed by the *destrInref* code afterwards.

The *value* of the reference function application is obtained by dereferencing the temporary result pointer *tmp*.

The $destr$ code consists of the right brace closing the scope of tmp. No destructor is called for tmp, since it is only a *reference* to part of some value destroyed elsewhere. Afterwards the value of the `inref` argument is destroyed by the $destrInref$ code. Again this is empty if `id` has no `inref` parameter. Also, if `id` was used as `inref` argument in some enclosing reference function application, the $destrInref$ code is not produced together with the $destr$ code, but only later on separately.

6.4.2 Conditional Expression

A conditional expression

$$\texttt{expr}_1 \ \texttt{?} \ \texttt{expr}_2 \ \texttt{:} \ \texttt{expr}_3$$

is treated like a (non-strict) reference function with two `inref` arguments $\texttt{expr}_2$ and $\texttt{expr}_2$, where only one of these is in fact evaluated, and the value of this one is used as result value. Which of the two arguments is evaluated is given by the value of the first argument $\texttt{expr}_1$. The code has the following form, assuming that the result type of the conditional expression, i.e., the type of $\texttt{expr}_2$ and $\texttt{expr}_3$, is T:

```
constrInref:
    { int _tmp;
      constr(t(expr1))
      _tmp = value(t(expr1));
      destr(t(expr1))
      if (!_tmp) goto label1;
        constrInref(expr2)
        goto label2;
      label1:
        constrInref(expr3)
      label2:;
constr:
    if !inref: constrInref(expr1 ? expr2 : expr3)
      { T^C *tmp;
        if (_tmp) {
          constr(expr2)
          tmp = &value(expr2);
          destr(expr2)
        } else {
          constr(expr3)
          tmp = &value(expr3);
          destr(expr3)
        }
value:
        *tmp
```

$destr$:
    }
  if `!inref`: $destrInref$(`expr`$_1$ `?` `expr`$_2$ `:` `expr`$_3$)
$destrInref$:
    `if (`$\_tmp$`) goto` $label_3$;
      $destrInref$(`expr`$_3$)
      `goto` $label_4$;
    $label_3$:
      $destrInref$(`expr`$_2$)
    $label_4$:;
  }

Again, the $constrInref$ code is emitted first. Like for a reference function it consists of the $constrInref$ code for the `inref` argument. To decide which of the two `inref` arguments, however, the value of the first argument is needed. It is constructed in a temporary variable "$\_tmp$" named like the temporary variable "tmp" for the result of the conditional expression, however with an underscore "_" prepended. The generated `if` command will execute either $constrInref$(`expr`$_2$) or $constrInref$(`expr`$_3$), depending on the value of "$\_tmp$". It is emitted in form of (conditional) `goto` commands, since $constrInref$(`expr`$_2$) and $constrInref$(`expr`$_3$) might contain opening braces which would destroy the structure of an ordinary `if` command.

The $constr$ code consists of the declaration of the temporary result pointer and its initialization with the address of the value of either `expr`$_2$ or `expr`$_2$, again depending on the value of "$\_tmp$". This time, an ordinary `if` command can be generated, since the braces in the braces in both branches are balanced.

The $destr$ code ends the scope of the result pointer. However, the value "$\_tmp$" of the condition is still needed in the $destrInref$ code. This consists of an `if` command for executing either $destrInref$(`expr`$_2$) or $destrInref$(`expr`$_3$), depending on "$\_tmp$". Again, the `if` command is in fact emitted in form of conditional `goto`s because of possible closing braces in its branches. At the very end of the $destrInref$ code, the scope of the temporary variable "$\_tmp$" holding the value of the condition is ended with a closing brace.

The condition actually used is not just "`expr`$_1$", but "`t(expr`$_1$`)`". The truth function `t` for the type of `expr`$_1$ is additionally applied, like for the conditions in Mona `if` commands and loops, to map the value to a Mona built-in `Bool` value, represented by a C++ `int`. The code generator determines the truth function to apply via an added type node representing the type of `expr`$_1$. An additional type node that represents the result type of the conditional expression is required. In case of a function application, the result type can be determined from the symbol representing the applied function. For a conditional expression this is not possible, thus additional type information is needed.

The labels $label_i$ used in the code are of the same form as the temporary variables: "$\_count$". This time the value of a global *label counter* is used as $count$ and incremented for each new label needed. Labels reside in a separate name space in C++. Their scope is the enclosing function, so the label counter is set to "0" each time before a procedure body is emitted.

If the result of the conditional expression is not used as `inref` argument, the value "$\_tmp$" of the condition would be tested twice in the generated $constr$ code as it is presented above: once in the $constrInref$ code, and once in the $constr$ code itself. This is not necessary if both parts of the code are emitted at once. In this case, the two `if`s can be combined. We actually emit the following code in this case:

$constr$:

```
{ int _tmp;
  constr(t(expr₁))
  _tmp = value(t(expr₁));
  destr(t(expr₁))
  { T^C *tmp;
    if (!_tmp) goto label₁;
      constrInref(expr₂)
      constr(expr₂)
      tmp = &value(expr₂);
      destr(expr₂)
      goto label₂;
    label₁:
      constrInref(expr₃)
      constr(expr₃)
      tmp = &value(expr₃);
      destr(expr₃)
    label₂:;
```

Implication, disjunction and conjunction expressions are interpreted as abbreviations for conditional expressions, namely:

- $expr_1$ `=>` $expr_2$ for $expr_1$ `?` $expr_2$ `:` t,
- $expr_1$ `||` $expr_2$ for $expr_1$ `?` t `:` $expr_2$,
- $expr_1$ `&&` $expr_2$ for $expr_1$ `?` $expr_2$ `:` f,

Code is generated for these conditional expressions. The symbols t and f are the *true value* and *false value* of the type of $expr_2$. They are specified for a type T with "`axiom type` T `trueval` t" and "`axiom type` T `falseval` f". The code generator obtains them via an added type node representing the type of $expr_2$, which is also the result type of the expressions. Furthermore, also the type of $expr_1$ is given by an added type node, so that the truth function to be applied in the code for the corresponding conditional expression can be determined.

6.4.3 Dereferencing

A dereferencing expression

$type$ `@` $expr$

is treated like a reference function without `inref` parameter. This results in the following code:

$constrInref$:

$constr$:

```
{ type^C *tmp;
  constr(a(expr))
  tmp = (type^C*)value(a(expr));
  destr(a(expr))
```

$value$:

```
  *tmp
```

$destr$:

```
}
```

$destrInref$:

No $constrInref$ or $destrInref$ code is emitted, since there is no `inref` parameter.

The value of the given address *expr* is mapped to the built-in `Address` type of Mona with the *address function* `a`. This function is specified for the type *T* of *expr* with "`axiom type` *type* `address a`". The code generator obtains it via an added type node representing the type of *expr*.

The value of the temporary result pointer for the dereferencing expression is obtained from the address value simply by a type cast.

6.4.4 Type Size and Alignment

A `sizeof` expression like

```
sizeof type
```

is treated like a parameterless function with the Mona built-in `Int` type as result type. The corresponding code is:

$constrInref$:

```
{ int tmp;
  tmp = sizeof(type^C);
```

$constr$:

```
if !inref: constrInref(sizeof type)
```

$value$:

```
  tmp
```

$destr$:

```
if !inref: destrInref(sizeof type)
```

$destrInref$:

```
}
```

Analogous code is generated for an `alignof` expression

```
alignof type
```

It uses the "`__alignof__`" function provided by the GNU compiler in the same way as the C `sizeof` function is used in the code above. The result of "`__alignof__(T)`" is the minimal alignment required for type `T`, i.e., the smallest number that must divide the address of any value of type `T`.

6.4.5 Representation

A representation expression

```
repr expr
```

yields the representation of the value of an expression *`expr`* of a `new` type. A `new` type is translated to the same C++ type as its representation type. Thus no code is generated for the `repr` operator, i.e., the code for "`repr` *`expr`*" is just the code for "*`expr`*":

$constrInref$:
  $constrInref$(*`expr`*)

$constr$:
  $constr$(*`expr`*)

$value$:
  $value$(*`expr`*)

$destr$:
  $destr$(*`expr`*)

$destrInref$:
  $destrInref$(*`expr`*)

6.4.6 Symbols

For a symbol *`id`* representing a variable or constant, no temporary variable is declared, constructed and destroyed. The C identifier representing the symbol in the code is simply used as value of the expression:

$constrInref$:

$constr$:

$value$: *`id`*$^{\mathsf{C}}$

$destr$:

$destrInref$:

6.5 Types

Type table entries can represent the type of any kind of Mona symbol. This includes of course function and procedure types. However, the types which have to be mapped to C++ types by the code generator can only be

- built-in types,
- `new` types,
- record types, or
- union types.

Built-in types have a simple correspondence to C++ types. The built-in `Int` type of Mona, e.g., is translated to a C++ `int`. See Appendix D for a complete list.

A `new` type entry in the type table contains a reference to the entry of the corresponding representation type. The code for a `new` type is just the code for this representation type.

The type table entry for a record or union type contains the corresponding list of components. Each component consists of its name and its type. A Mona record type is translated to a C++ `struct` with corresponding components. A Mona union type is analogously translated to a C++ `union`. However, these types are treated differently in C++ than in Mona. In Mona, types are considered to be equal if their structure is the same. This means that two record or union types are equal if their component lists are equal. In contrast to this, two C++ `struct` or `union` types are considered to be different except if they have the same name. In particular, unnamed `struct` or `union` types are always different from all other types. Thus it makes no sense to use, e.g., an unnamed `struct` type in a parameter declaration, since no argument can ever be of this type. The `struct` type must be named, so that also values of this type can be defined.

Therefore, names are introduced by the code generator for record and union types. When code for a record or union type has to be emitted, it is checked whether the corresponding type table entry is already named. If not, a type name is chosen, a `struct` or `union` definition for this name is written to the *structure definitions file*, and the name is stored in the type table at the corresponding entry. Only this type name is then emitted as code for this entry.

The structure definitions file has to be included in the module header file before the list of prototypes, since type names will in general already be used in prototypes. The chosen type names must therefore be unique in the whole repository to prevent collisions that might appear when the module is used, together with others, as argument in a new instance of some functor. They are built, similarly to the unique names chosen for global symbols, by combining a type number with the unique module name described in the following section.

Though unique names can be chosen locally, it is not possible to keep a one-to-one correspondence between names and types without maintaining a global database of all record and union types ever used in any of the modules in the repository. Thus we allow different names to be chosen for equal types in different modules. No attempt is made, in fact, to use the structure and union definitions provided by a module anywhere else. Every module uses its own type names. This requires, however, type casts to be applied to arguments of entities provided by parameter modules. We therefore generate for each entity provided in form of a C++ procedure by some module an additional preprocessor macro that expands to a call of the procedure with arguments casted to the expected types. This macro gets actually the unique name chosen for the entity. The procedure name gets an additional underscore "_" prepended. Instead of a procedure prototype

$$\texttt{void } id^{C}(type_1^{C}\ \&id_1^{C},\ type_2^{C}\ \texttt{const}\ \&id_2^{C},\ \ldots);$$

the following is thus written to the header file:

$$\begin{array}{l}\texttt{void \_}id^{C}(type_1^{C}\ \&id_1^{C},\ type_2^{C}\ \texttt{const}\ \&id_2^{C},\ \ldots);\\ \texttt{\#define } id^{C}(id_1^{C},\ id_2^{C},\ \ldots)\ \backslash\\ \qquad \texttt{\_}id^{C}((type_1^{C}\&)(id_1^{C}),\ (type_2^{C}\ \texttt{const}\&)(id_2^{C}),\ \ldots);\end{array}$$

The procedure definition in the source file gets correspondingly the name "$\texttt{\_}id^{C}$" instead of "id^{C}". Actually, the type casts in the macro definition are only emitted where necessary: for record and union parameter types and for `new` parameter types whose representation requires a cast.

For an `extern` definition, the type casts are added to the macro definition that is anyways generated (cf. Subsection 6.1.10). For an `extern` *function*, also a cast for the variable receiving the result value is added if necessary. For

$$\texttt{fun } id(id_1\texttt{: } type_1,\ \ldots)\texttt{: } type\ \texttt{extern}\ extern\text{-}id$$

this takes the form

$$\begin{array}{l}type^{C}\ extern\text{-}id(type_1^{C}\ id_1^{C},\ \ldots);\\ \texttt{\#define } id^{C}(\texttt{\_},\ id_1^{C},\ \ldots)\ \backslash\\ \qquad ((type^{C}\&)(\texttt{\_}) = extern\text{-}id((type_1^{C}\&)(id_1^{C}),\ \ldots))\end{array}$$

Parameters and result variable are casted to *reference* types. A C++ reference to a `struct` type can be casted to a reference to another `struct` type. It is, however, impossible to cast a `struct` itself to another `struct` type.

No type casts are necessary for `alias` or `builtin` definitions. For an alias, the casts will be done by the defining entity. The only built-in operations on values of a record or union type are the corresponding default constructors and destructors and the component selectors. Constructors and destructors are empty, i.e., do not use their argument. The component selectors work on any C++ `struct` or `union` that has a component of the given name.

6.6 Symbols

The symbols representing identifiers in the annotated syntax tree are translated to C identifiers. The translation of the *global* symbols, i.e., of the symbols for which externally visible code is produced, is different from the translation of symbols representing parameters or local constants or variables.

6.6.1 External Entities

The external name chosen for a *global* symbol must uniquely identify the entity represented by this symbol within the whole package hierarchy stored in the central repository. A unique name for the module containing the symbol is more or less provided by the way the repository is organized. It is not easily possible to associate modules with `module` units. As discussed in Section 5.3, a single `module` unit usually produces several 'unnamed' modules. Such an 'unnamed' module will probably soon be used as argument in some other functor application. Then it is no longer clear which `module` unit actually produced the 'unnamed' module. Therefore, each module is associated with the *functor* it is an instances of, i.e., the code for the module is stored in a sub-directory of the functor's directory in the repository. Different instances of a functor simply get increasing numbers. For this purpose, an *instance counter* is maintained for every functor in the repository. Thus the global identifier of a functor, together with an instance number, uniquely identifies a certain module in the repository. A C identifier $module\text{-}name$ of the form

$$P_1\_\cdots\_P_k\_F\_i$$

can therefore be used to uniquely determine the ith instance of a functor named P_1<>···<>P_k<>F. Since underscore characters and digits are not allowed in functor and package names (cf. the definition of the `unitId` terminal class in Appendix B), this cannot lead to collisions with names chosen in the same way for other modules.

It remains to uniquely identify the symbols contained in a module. The symbol names do not suffice for this purpose. On the one hand side, Mona distinguishes several *name spaces*, namely for package names, specification names, functor names, module names, type names and value names. The last one comprises names of procedures, functions, constants and variables. Of relevance for the code generator is that a type may have the same name as, e.g., a function. Moreover, each type provides separate name spaces for the names of its constructors and destructors. On the other hand, also *overloading* of function or procedure names, based on different parameter types, is allowed. Thus two symbols representing different functions may have the same name. Therefore we decided to use numbers also for identifying the symbols in a specific functor instance, like for identifying the instances of a certain functor. The symbols to be marked are simply numbered, starting from 0, in the order they are found when traversing the annotated syntax tree. The C identifier

$$\_s\_module\text{-}name$$

is then used as unique name for the symbol with number s of the module with unique name $module\text{-}name$. This yields external names starting with an underscore followed by a digit, which are rather unlikely to be used elsewhere.

6.6.2 Parameters and Local Values

For *local* symbols, i.e., symbols representing function or procedure parameters, or constants or variables defined locally to command blocks, C identifiers more similar to the corresponding Mona identifiers can be chosen. It is, however, not possible to use the Mona identifiers directly. Consider the following Mona definitions:

```
var v: T  extern e

proc p(inout e: U) {
  ... v ...
}
```

The external variable e is used with its Mona name v in the body of the procedure p. The code produced for this procedure body will, after preprocessing, refer directly to e. If now the parameter of p would not be renamed, this e in the generated procedure body would refer to the *parameter* instead of to the external variable. The same problem appears also when e is used as local constant or variable in the body of the Mona procedure, instead of as parameter. Therefore we prepend an underscore "_" to the Mona names for parameters and local constants/variables. The resulting identifiers cannot collide with any Mona identifiers, since these are not allowed to have the underscore as first character (cf. Appendix B).

Furthermore, C representations of the special kinds of Mona identifiers have to be chosen. We do this as follows:

- Operator identifiers "`operator` op" are translated to C identifiers of the form "`__op_`op^{C}", where op^{C} is a name for the specific operator op. The operator name "operator !"', e.g., is translated to "`__op_excl`". If op itself is allowed in a C identifier, it is used as op^{C}. The operator name "operator 0f"', e.g., is translated to "`__op_0f`".
- Character identifiers "`'`chr`'`" are translated to C identifiers of the form "`__chr_`chr^{C}", where chr^{C} is a name for the specific character chr. The character identifier "`'!'`", e.g., is translated to "`__chr_excl`". If chr itself is allowed in a C identifier, it is used as chr^{C}. The character identifier "`'a'`", e.g., is translated to "`__chr_a`".
- Digit identifiers "`#`dig" are translated to C identifiers of the form "`__dig_`dig". The digit identifier "#0", e.g., is translated to "`__dig_0`".
- The identifier "`''`" is translated to the C identifier "`__str_nil`".

All the generated identifiers start with two underscores to avoid collisions with valid Mona identifiers like "`dig_0`", as well as with C identifiers like "`_dig_0`" resulting from using such Mona names for parameters or local variables.

Chapter 7

Special Code Generator Features

In this chapter, we describe how the Mona code generator exploits information provided by module specifications. Furthermore, we describe how the code generator chooses parameter passing mechanisms depending on the mode and the type of parameters. In contrast to what was said about parameter passing in the previous chapter (Section 6.2), the code generator uses in fact 'call by value' in certain cases for efficiency reasons. It has the flexibility to choose parameter passing mechanisms as needed, because these are not fixed by the specification of the semantics of the parameter modes in Mona.

After that, some further cases are discussed in which actually better code is generated than described in the previous chapter. Finally we describe how debugging information in form of `#line` directives referring to Mona sources is added to the code. This allows to trace execution of Mona source code with usual source level debuggers.

7.1 Using Specifications

Part of the information provided in Mona module specifications is (going to be) exploited by the code generator:

- Executable specifications of function result values are translated into default implementations of the specified functions on request.
- Executable input/output specifications of functions, procedures, constructors and destructors are translated into runtime checks of the input and output values of the specified entities.

The latter is not yet implemented, so we only sketch below what we plan to do.

7.1.1 Default Implementations

A function can be defined as `default`. This means that a default implementation should be generated from the *specification* of the function. The type-checker requires for this that the specification is of the form

```
fun id(param, ...): type = expr
```

and checks whether the expression *expr* specifying the result value of the function is *executable*. Currently, the code generator requires *expr* to be a core language expression as it is allowed in definitions. This will be relaxed in the future so that also a restricted form of the `let` blocks allowed in specification expressions can be used.

An annotated syntax tree for the expression *expr* is generated by the type checker in case the function *id* is defined with `default` in a functor implementing the above specification. This tree is stored as implementation information for the symbol representing the function. When the functor is instantiated, code can be generated for this annotated tree like for any expression tree occurring directly in the module tree. The code generator produces a procedure definition of the following form from the expression:

```
inline void id^C(type^C &_, param^C, ...)
{
  constr(expr)
  cc^C(_, value(expr));
  destr(expr)
  return;
}
```

The expressions specifying result values of functions defined with `default` are expected to be simple. Thus `inline` definitions are generated for such functions.

The code for the procedure body sets the result parameter "_" to the value of the expression with a call of the procedure representing the copy constructor *cc* for the result type *type*. The resulting code is the same as for a function definition of the form

```
fun id(param, ...): type inline return expr
```

7.1.2 Checking Axioms

Debugging code for a module should contain code that checks at run-time for each function or procedure call whether the specified input condition holds before the call and whether the corresponding output condition holds afterwards. Analogously, the conditions specified for constants or variables defined globally in a module should be checked after initialization of the module. This is not yet implemented, but we plan to provide additional preprocessor macros that wrap the actual C++ procedure calls with code checking the input and output conditions.

Each procedure, function, constructor and destructor symbol contains the specified input and output conditions in form of boolean expressions. Function symbols may have an additional result expression, as given by specifications of the form

```
fun id(param, ...): type = expr
```

Such specifications are considered as abbreviations for

```
fun id(param, ...)r: type
  output e(r, expr)
```

The check of the result expression is thus part of the output condition. The function `e` is the *equality* of the function result type. It is specified as "`axiom type` *type* `equality e`".

The input/output specification of a procedure or function can be further restricted with corresponding *axioms* like follows:

```
proc id(param, ...)
  input  i1
  output o1
axiom proc id(param, ...)
  input  i2
  output o2
```

Thus sets of input/output condition pairs can be attached to symbols. The resulting combined input condition for the above example is "i_1 `||` i_2", and the combined output condition is "(old i_1 => o_1) && (old i_2 => o_2)".

The `old` operator allowed in output conditions refers to the values of `inout` parameters and global variables at *call* time. Thus, all `old` expressions occurring in the output condition(s) have to be evaluated before the actual procedure call, and their values have to be saved for use in the evaluation of the output condition after the call.

Finally, the appropriate *truth* functions have to be applied to all conditions to map their results to the built-in `Bool` type of Mona.

7.2 Parameter Passing

An interesting feature of Mona is that the semantics of the parameter modes `in`, `inref`, `inout` and `out` are not specified in terms of certain parameter passing mechanisms. This allows suitable parameter passing mechanisms to be chosen as needed by the code generator. It allows even different mechanisms to be chosen, depending on the parameter type, for implementing a certain parameter mode. This is indeed exploited by the Mona code generator.

7.2.1 Input Parameters

In Chapter 6 we stated that in Mona all parameters are passed by reference. This is actually not the case. It would make it impossible to pass parameters in registers. For `inout` and `out` parameters there is actually no other choice than passing references. Their meaning is that changes of the formal parameters are reflected in the actual parameters. A copy-in/copy-out scheme might be more efficient than passing a reference in some cases, but this is not supported by C or C++. However, `in` parameters may well be passed 'by value'. This can be done without the need for calls of copy constructors to initialize the formal parameters. In C++, this is necessary, because formal value parameters may be changed without effect on the actual parameters. Therefore, they are treated like local variables that are initialized with the values of the corresponding actual parameters by means of copy constructors and are destroyed at the end of the defining body by the appropriate destructor. In Mona, however, the formal `in` parameters are *constants*, i.e., cannot be changed. Thus, as well as references can be passed, also *shallow copies* of the actual parameters, made without copy constructors and without the need of calling destructors at the end of the body, can be passed without problems. This is still a kind of 'call by reference'. If the parameter type is, e.g., some `new` type represented by a built-in `Address` of some heap storage region, only this `Address`, i.e., a reference to the actual data, is copied. Also other built-in types might refer to data stored elsewhere. Therefore the same restrictions w.r.t. supplying arguments apply as when 'real' references are used: actual `in` parameters may not occur as (parts of) actual `inout` or `out` parameters (and vice versa) in the same call, i.e., aliasing via such parameters is not allowed.

Using C++ value parameters instead of references makes only sense if there is a chance that the actual parameter is kept in a register. We assume that this is the case for all built-in Mona types. The meaning of these types is defined in terms of the corresponding C types. The size of C types, in turn, is architecture-dependent. So the above assumption seems to be a good guess. We assume further that Mona record or union types, i.e., C `struct` and `union` types, are never kept in registers. This might be wrong, e.g. for records consisting of only a single integer component. However, `structs` or `unions` will always be forced into storage by the necessary parameter type casts done in the macro wrappers for all procedures in the code (cf. Section 6.5).

Thus a Mona input parameter declaration is translated as follows

$$\texttt{in}\ \mathit{id}\texttt{:}\ \mathit{type} \quad \longrightarrow \quad \mathit{type}^{\mathsf{C}}\ \mathit{id}^{\mathsf{C}}$$

if $\mathit{type}^{\mathsf{C}}$ is neither a `struct` nor a `union` type.

7.2.2 Inref Parameters

As `in` parameters, also `inref` parameters can not be changed in the body of a reference function. However, in contrast to `in` parameters, they are allowed to be returned as result of a reference function. And, furthermore, changes of the result of the reference function are reflected in the supplied `inref` argument.

For the same reason as for `in` parameters, it makes also sense to pass `inref` arguments by value. Consider, e.g, an `Array` type implemented as a built-in `Address` of some heap storage region. For a reference function

```
reffun operator[](inref a: Array, i: Index): Elem
```

providing access to array elements, it suffices if the address `a` is passed by value. However, care has to be taken in cases like the following:

```
reffun operator +(inref i: Int): Int  return i
```

This implements an identity function whose result may be changed. However, changes of the result should also change the argument. This requires that in this case the `inref` argument is passed by reference. In general, this is required if the result of the reference function is a reference to the `inref` argument itself. This is, e.g., also the case in the following example:

```
type N new R
reffun representation(inref n: N): R  return repr n
```

For record types, also returning a component of the record would pose the same problem. However, as discussed in the previous subsection, it anyways makes no sense to pass record types by value in the code for Mona.

We do not check the body of a reference function to determine whether a reference to the `inref` argument might be returned. Instead, we check whether this is *possible* by checking whether the type of the `inref` parameter can be casted to the result type. The only type cast provided by Mona is the `repr` operator. Thus a type `T` can be casted to a type `R` only if either `T` and `R` are equal or `T` is a `new` type and its representation type can be casted to `R`.

A Mona `inref` parameter declaration is hence translated as follows

$$\texttt{inref}\ \mathit{id}\texttt{:}\ \mathit{type} \quad\longrightarrow\quad \mathit{type}^{\mathsf{C}}\ \mathit{id}^{\mathsf{C}}$$

if $\mathit{type}^{\mathsf{C}}$ is neither a `struct` nor a `union` type, and if *type* cannot be casted to the result type of the corresponding reference function.

7.2.3 Result Values

Translating all Mona functions to C++ procedures poses a similar problem as using only reference parameters. The result can never be passed in a register. Therefore we translate in fact Mona functions to C++ functions if their C++ result type is no `struct` or `union`. However, we keep the procedural interface for the corresponding type-casting macros. Thus when a Mona module is used from an external C program, still all functions are available only as procedures.

For a Mona function definition of the form

$$\texttt{fun}\ \mathit{id}\texttt{(}\mathit{param}\texttt{, ...):}\ \mathit{type}\ \mathit{cmd}$$

this means that a C++ function definition like follows is generated if $type^C$ is no `struct` or `union`:

```
typeᶜ idᶜ(paramᶜ, ...)
{ typeᶜ _;
  cmdᶜ
}
```

The result identifier "`_`" used in the code for the `return` command is declared as a local variable in the function body. It must now of course be issued as result expression, i.e., "`return _`" has to be used instead of just "`return`" in the code for a Mona "`return` *expr*" inside such a function body (see Subsection 6.3.6 in the previous chapter).

The definition of the corresponding type-casting macro takes then the form

```
#define idᶜ(_, ...) ((_) = _idᶜ(...))
```

Note that no type cast is required for the result identifier "`_`", since its type is neither a `struct` nor a `union` (cf. Section 6.5).

A Mona function definition with an explicitly named result value

```
fun id1(param, ...) id2: type cmd
```

is analogously translated like follows if $type^C$ is no `struct` or `union`:

```
typeᶜ id1ᶜ(paramᶜ, ...)
{ typeᶜ id2ᶜ;
  cmdᶜ
  return id2ᶜ;
}
```

A return command returning the locally declared result identifier id_2^C has to be inserted at the end of the body, because for this type of function definition the Mona body need not contain a return command. The result of the function must be constructed in id_2^C in the body of the function definition. In case of a Mona return command occurring in the body, of course "`return` id_2^C" has to be used instead of only "`return`" in the corresponding C++ code.

Again, the type-casting macro provides a procedural interface:

```
#define id1ᶜ(id2ᶜ, ...) ((id2ᶜ) = _id1ᶜ(...))
```

Such a function with named return value behaves exactly like a constructor: the named value is constructed in the body. Indeed we generate exactly the same code for a constructor definition of the following form, again if $type^C$ is neither a `struct` nor a `union`:

```
constr id2: type'id1(param, ...) cmd
```

Finally, we come to reference functions. Their result is always a C++ pointer and should thus be passed in a register. A reference function definition

```
reffun id(param, ...): type cmd
```

is therefore in fact always translated to a C++ function of the form

```
typeC *idC(paramC, ...)
{ typeC *_;
  cmdC
}
```

Again, "`return _`" must now be issued instead of just "`return`" in the code for a Mona "`return` *expr*" inside a reference function body.

Since no destructors have to be called for the results of reference functions, it is no problem if they are used as functions also in external C or C++ programs. We therefore decided to make an exception to the strategy of providing only procedures: reference functions are provided as functions, with type-casting macros of the following form:

```
#define idC(...) (*_idC(...))
```

The dereferencing of the function's result in the macro definition allows external usage of the function without the necessity to explicitly deal with pointers. The macro behaves like a C++ function with a reference type as result type.

Providing reference functions as functions instead of as procedures requires finally to change the code for reference function applications presented in Chapter 6. Instead of

```
if !inref: constrInref(id(expr, ...))
{ TC *tmp;
  constr(expr), ...
  idC(tmp, value(expr), ...);
  ..., destr(expr)
```

the following is emitted as *constr*(`id(expr, ...)`) for a reference function *id* with result type *T*:

```
if !inref: constrInref(id(expr, ...))
{ TC *tmp;
  constr(expr), ...
  tmp = (TC*)&idC(value(expr), ...);
  ..., destr(expr)
```

The type cast "$(T^{\mathsf{C}}*)$" for the address of the function result is only necessary if T^{C} is a `struct` or `union` type.

7.3 Further Optimizations

Besides the optimization of parameter passing described in the previous section, also in some other cases better code is emitted than described in Chapter 6. These cases, together with the corresponding optimized code, are described in this section.

7.3.1 Conditional Expressions

In Subsection 6.4.2 in the previous chapter, we said that code for a conditional expression is produced like for a reference function with two `inref` parameters. Depending on the value of the condition, one of the `inref` arguments is evaluated, and a reference to its value is returned as result. However, the Mona type-checker treats the result of a conditional expression never as variable, i.e., it does not allow to change the result of a conditional expression even if both branches are changeable. Thus it is as well possible to return a 'shallow copy' of the evaluated branch. This means that, in the *constr* code for the conditional expression, the *value* of the evaluated branch, instead of its address, is assigned (not copied!) to the temporary result variable. The code generator indeed produces code like this if the value of one of the two branches might reside in a register. The code emitted as $constr(\mathtt{expr}_1\ \texttt{?}\ \mathtt{expr}_2\ \texttt{:}\ \mathtt{expr}_3)$ in this case is thus in fact:

```
if !inref: constrInref(expr₁ ? expr₂ : expr₃)
{ Tᶜ tmp;
  if (_tmp) {
    constr(expr₂)
    tmp = value(expr₂);
    destr(expr₂)
  } else {
    constr(expr₃)
    tmp = value(expr₃);
    destr(expr₃)
  }
```

Correspondingly, the *value* of the conditional expression is given by *tmp*, instead of by **tmp*, in this case.

7.3.2 Copy Constructor Calls

In the code described in Chapter 6 for a "`return` *expr*" command occurring in a Mona function body, the result parameter "_" is constructed with a call of the procedure representing the copy constructor *cc* for the result type of the function:

```
constr(expr)
ccᶜ(_, value(expr));
destr(expr)
```

If the result expression yields only a temporary value, as does, e.g., a function application, this means that this temporary value is constructed in a temporary variable, then it is copied, and afterwards the value of the temporary variable is destroyed. The call of the copy constructor, as well as the call of the default destructor for the temporary variable, can be avoided by constructing the temporary value directly in the result parameter. The same holds for copy constructors called to initialize local values, and also if a copy constructor is called explicitly.

We therefore produce in fact code $constr($_, *expr*$)$ constructing the value of *expr* directly in "_" instead of the copy constructor call in the code for "`return` *expr*", and similarly $constr($`id`, *expr*$)$ instead of the copy constructor call in the code for a local constant or variable definition of the form

```
var id: type`cc(expr)
```

Finally, we generate

$constr(\texttt{expr}_1)$
$constr(\texttt{expr}_1,\ \texttt{expr}_2)$
$destr(\texttt{expr}_1)$

as code for an explicit copy constructor call like

```
constr expr1: type`cc(expr2)
```

The code actually emitted as $constr(\texttt{expr}_1,\ \texttt{expr}_2)$ is determined by $\texttt{expr}_2$. If $\texttt{expr}_2$ is a reference function application, a dereferencing expression or simply a symbol representing some constant or variable, then $constr(\texttt{expr}_1,\ \texttt{expr}_2)$ is in fact nothing else than the call of the copy constructor *cc* for the type of $\texttt{expr}_1$:

$constr(\texttt{expr}_2)$
$\texttt{cc}^{\mathsf{C}}(value(\texttt{expr}_1),\ value(\texttt{expr}_2));$
$destr(\texttt{expr}_2)$

Note, however, that in all these cases actually no destructor call is contained in $destr(\texttt{expr}_2)$ (cf. Section 6.4).

If $\texttt{expr}_2$ is a function application, the copy constructor call is not necessary. The following is emitted as $constr(\texttt{expr}_1,\ \texttt{id}(\texttt{expr}_2,\ \ldots))$:

$constr(\texttt{expr}_2),\ \ldots$
$\texttt{id}^{\mathsf{C}}(value(\texttt{expr}_1),\ value(\texttt{expr}_2),\ \ldots);$
$\ldots,\ destr(\texttt{expr}_2)$

The value of $\texttt{expr}_1$ (which must be a reference) is directly supplied as result parameter in the call of the procedure $\texttt{id}^{\mathsf{C}}$ representing the function *id*.

For conditional expressions, it depends on the two branches whether a copy constructor call is necessary or not. The following code is emitted as $constr(\texttt{expr}_1,\ \texttt{expr}_2\ ?\ \texttt{expr}_3\ :\ \texttt{expr}_4)$:

```
if (({int __;
      constr(t(expr_2))
      __ = value(t(expr_2));
      destr(t(expr_2))
      __;})) {
  constr(expr_1, expr_3)
} else {
  constr(expr_1, expr_4)
}
```

This constructs the result of either $expr_3$ or $expr_4$ in the value of $expr_1$, depending on the value of the condition $expr_2$. The symbol *t* represents the truth function for the type of $expr_2$ that has to be applied to the condition.

For `sizeof` and `alignof` expressions, it makes in fact not much difference whether the copy constructor is called or not. The following is emitted as *constr*(*expr*, `sizeof` *type*):

```
value(expr) = sizeof(type^C);
```

The analogous code is emitted as *constr*(*expr*, `alignof` *type*): `sizeof` is replaced by `__alignof__`.

For the `repr` operator, finally, no code is produced. Thus *constr*($expr_1$, `repr` $expr_2$) is in fact the same as *constr*($expr_1$, $expr_2$).

7.3.3 Built-in Operations

The preprocessor macro definitions generated for `builtin` definitions are actually not used in the generated code. The code for built-in operations is directly inlined, and some optimizations are applied: For the identity functions for most built-in types, e.g., no code is in fact generated. Consider the following Mona definitions:

```
type Int  builtin Int

fun identity(i: Int): Int  builtin Int
```

For the defined identity function on built-in integers, the following macro definition will be emitted to the header file (cf. Appendix D):

```
#define identity^C(_, i^C) ((_) = (i^C))
```

For an application "`identity(`*expr*`)`" of the identity function, however, no temporary variable declaration with initialization by a call of the `identity`^{C} macro is emitted, but only the code for the argument *expr*.

Furthermore, for empty default constructors and destructors for built-in types, also no code for evaluating the argument to construct or destroy, which might be the result of some reference function application, is emitted.

Finally, some attempts are made to produce better code for expressions representing literals of built-in types. In particular the built-in `String` operations supporting string and floating point number literals produce quite inefficient code. A string literal is translated by the parser into a nested application of a function that combines a single character with a string to a new string that is one character longer (cf. Chapter 4). The built-in function that implements this for built-in characters and built-in strings allocates a new string, one character longer than the argument string, assigns the character to the first position in the new string and copies the argument string into the rest of the new one. Using this as is would imply that for every single character in a string literal a new string had to be allocated temporarily, just to be freed again after its contents was copied into the next longer string. We optimize this by first examining the whole expression tree representing the literal to determine the length of the string, so that code for only a single allocation is needed.

We intend to invest some more work in optimizations of built-in operations. In particular, we plan to implement compile-time evaluation of constant expressions in the Mona code generator. This would, besides further optimizing literals of built-in types, also allow to at least partially translate Mona `switch` commands to C `switch` commands, instead of to `if` cascades as described in Subsection 6.3.3 of the previous chapter.

7.4 Debugging Information

The code generator adds `#line` preprocessor directives to the C++ source and inline definitions file which refer to a pretty printed version of the Mona source for the instantiated functor. Thus, when a source level debugger like `gdb` is used to trace the execution of a program using a Mona module, the Mona source will be displayed when an entity provided by the module is executed.

The file referred to by the `#line` directives is generated from the syntax tree for the instantiated functor with the Mona pretty printer and kept together with the tree in the repository, if debugging code is requested. The line information is taken from the *`lineNr`* nodes in the annotated syntax tree that are generated by the pretty printer (cf. Section 2.4). In between the `#line` directives, the code generator formats the code corresponding to the formatting of the Mona functor source by the pretty printer. This means that the generated C++ code contains line breaks only at positions where it is sure that also the pretty printer outputs a line break in the corresponding position in the Mona source.

The execution of Mona code can thus be conveniently traced. However, access to values used in the Mona source or execution of functions or procedures still requires detailed knowledge of the C++ code produced by the code generator, in particular of the C identifiers that represent the Mona items.

APPENDICES

Appendix A

Grammar

In this appendix the complete (except for comments) grammar of Mona is given in an EBNF close to the one proposed by Wirth in [Wir77] and used, e.g., in [Wir85]:

- '*nonterminal*' denotes a nonterminal symbol,
- '`keyword`' denotes a keyword terminal symbol,
- other terminal symbols are surrounded by double quotes, like ' "?" ',
- '*`class`*' denotes a terminal class (defined in Appendix B),
- the vertical bar '|' separates alternatives (binds looser than concatenation),
- the parentheses '(' and ')' are used for grouping,
- the brackets '[' and ']' enclose optional items,
- the braces '{' and '}' enclose items that may occur zero or more times,
- ':=' separates the left and right hand sides of productions, and
- every production is explicitly terminated by a full stop '.'.

Though comments are part of the parser syntax in Mona, they are left out from the functor and core language grammar sections to keep the presented grammars more readable, and are treated separately in Section A.3.

The presented grammar shows an easy form of context-sensitivity: the rules for *moduleId*, *functorExpr* and *extPrefixExpr* (Subsection A.1.7) depend on the kind of the *unit* in which these nonterminals occur. It could of course be rewritten to a really context-free grammar by providing separate nonterminal symbols and separate rule-versions for all involved syntax domains. However, we think that the presentation as given is clearer, since it keeps single rules for equivalently structured domains, and pinpoints the differences where they occur.

A.1 Functor Language

$unit$:=
    $specification$
  | $functor$
  | $module$.

A.1.1 Specifications

$specification$:=
    `spec` $globalId$ ["`[`" [$moduleParams$] "`]`"]
        $spec$.

$spec$:=
    $specSeq$
  | $specBlock$
  | $include$
  | $functorUse$
  | $moduleSpec$
  | $import$
  | $typeSpec$
  | $constrSpec$
  | $destrSpec$
  | $procSpec$
  | $funSpec$
  | $reffunSpec$
  | $valSpec$
  | $varSpec$
  | $moduleAxiom$
  | $typeAxiom$
  | $constrAxiom$
  | $destrAxiom$
  | $procAxiom$
  | $funAxiom$
  | $reffunAxiom$
  | $valAxiom$
  | $varAxiom$
  | $generalAxiom$.

$specSeq$:= "`{`" {$spec$} "`}`" ["`;`"].

$specBlock$:= `let` $spec$ `in` $spec$.

$include$:= `include` $specExpr$ [(`only` | `except`) ids] ["`;`"].

$functorUse$: see Functors.

$moduleSpec$:=
    `module` id "`:`" $specExpr$
        ["`=`" $moduleExpr$] ["`;`"].

$import$: see Functors.

```
typeSpec :=
    type id
        [ “=” typeExpr [“;”]
        | “!=” typeExpr [“;”]
        | builtin id [“;”]
        ].

constrSpec :=
    constr id “:” typeExpr [“`” id] “(” [valueParams] “)” [“;”]
        [input valueExpr [“;”]]
        [output valueExpr [“;”]].

destrSpec :=
    destr id “:” typeExpr [“`” id] “(” [valueParams] “)” [“;”]
        [input valueExpr [“;”]]
        [output valueExpr [“;”]].

procSpec :=
    proc id “(” [valueParams] “)” [“;”]
        [input valueExpr [“;”]]
        [output valueExpr [“;”]].

funSpec :=
    fun id “(” [valueParams] “)” [id] “:” typeExpr
        [“=” valueExpr] [“;”]
        [input valueExpr [“;”]]
        [output valueExpr [“;”]].

reffunSpec :=
    reffun id “(” [valueParams] “)” [id] “:” typeExpr
        [“=” valueExpr] [“;”]
        [input valueExpr [“;”]]
        [output valueExpr [“;”]].

valSpec :=
    val id “:” typeExpr
        [“=” valueExpr] [“;”]
        [output valueExpr [“;”]].

varSpec :=
    var id “:” typeExpr
        [“=” valueExpr] [“;”]
        [output valueExpr [“;”]].

moduleAxiom :=
    axiom module moduleExpr
        “=” moduleExpr [“;”].
```

typeAxiom :=
  `axiom` `type` *typeExpr*
    (“=” *typeExpr* [“;”]
    | “!=” *typeExpr* [“;”]
    | `equality` *longId* [“;”]
    | `truth` *longId* [“;”]
    | `case` *longId* [“;”]
    | `address` *longId* [“;”]
    | `trueval` *longId* [“;”]
    | `falseval` *longId* [“;”]
    | `constructed` {*classifiedConstr*} [*defaultConstr*] [“;”]
).

constrAxiom :=
  `axiom` `constr` *id* “:” *typeExpr* [“‘” *id*] “(” [*valueParams*] “)” [“;”]
    [`input` *valueExpr* [“;”]]
    [`output` *valueExpr* [“;”]].

destrAxiom :=
  `axiom` `destr` *id* “:” *typeExpr* [“‘” *id*] “(” [*valueParams*] “)” [“;”]
    [`input` *valueExpr* [“;”]]
    [`output` *valueExpr* [“;”]].

procAxiom :=
  `axiom` `proc` *longId* “(” [*valueParams*] “)” [“;”]
    [`input` *valueExpr* [“;”]]
    [`output` *valueExpr* [“;”]].

funAxiom :=
  `axiom` `fun` *longId* “(” [*valueParams*] “)” [*id*] “:” *typeExpr*
    [“=” *valueExpr*] [“;”]
    [`input` *valueExpr* [“;”]]
    [`output` *valueExpr* [“;”]].

reffunAxiom :=
  `axiom` `reffun` *longId* “(” [*valueParams*] “)” [*id*] “:” *typeExpr*
    [“=” *valueExpr*] [“;”]
    [`input` *valueExpr* [“;”]]
    [`output` *valueExpr* [“;”]].

valAxiom :=
  `axiom` `val` *longId* “:” *typeExpr*
    [“=” *valueExpr*] [“;”]
    [`output` *valueExpr* [“;”]].

varAxiom :=
  `axiom` `var` *longId* “:” *typeExpr*
    [“=” *valueExpr*] [“;”]
    [`output` *valueExpr* [“;”]].

generalAxiom := `axiom` *valueExpr* [“;”].

classifiedConstr := *longId* "(" *constr* ["(" *longIds* ")"] ")" [","].

defaultConstr := `default` "(" *constr* ["(" *longIds* ")"] ")" [","].

constr := [*longId* "`=>`"] *longId*.

A.1.2 Functors

functor :=
    `functor` *globalId* ["`[`" [*moduleParams*] "`]`"] "`:`" *specExpr*
        def.

def :=
    defSeq
  | *defBlock*
  | *functorUse*
  | *moduleDef*
  | *import*
  | *typeDef*
  | *constrDef*
  | *destrDef*
  | *procDef*
  | *funDef*
  | *reffunDef*
  | *valDef*
  | *varDef*
  | *initDef*
  | *termDef*.

defSeq := "`{`" {*def*} "`}`" ["`;`"].

defBlock := `let` *def* `in` *def*.

functorUse :=
    `functor` *id* ["`[`" [*moduleParams*] "`]`"] "`:`" *specExpr*
        "`=`" *globalId* ["`;`"].

moduleDef :=
    `module` *id* "`:`" *specExpr*
        ("`=`" *moduleExpr* ["`;`"]
        | `forward` ["`;`"]
).

import := `import` *moduleExpr* [(`only` | `except`) *ids*] ["`;`"].

typeDef :=
    `type` *id*
        ("`=`" *typeExpr* ["`;`"]
        | `new` *typeExpr* ["`;`"]
        | `builtin` *id* ["`;`"]
        | `forward` ["`;`"]
).

```
constrDef :=
    constr id ":" typeExpr ["‘" id] "(" [valueParams] ")"
        ( [inline] cmd
        | alias typeExpr ["‘" id] ["(" [paramCasts] ")"] [";"]
        | forward [";"]
        ).

destrDef :=
    destr id ":" typeExpr ["‘" id] "(" [valueParams] ")"
        ( [inline] cmd
        | alias typeExpr ["‘" id] ["(" [paramCasts] ")"] [";"]
        | forward [";"]
        ).

procDef :=
    proc id "(" [valueParams] ")"
        ( [inline] cmd
        | alias longId ["(" [paramCasts] ")"] [";"]
        | extern id ["(" [paramModes] ")"] [";"]
        | builtin id [";"]
        | forward [";"]
        ).

funDef :=
    fun id "(" [valueParams] ")" [id] ":" typeExpr
        ( [inline] cmd
        | alias longId ["(" [paramCasts] ")"] [";"]
        | extern id ["(" [paramModes] ")"] [";"]
        | builtin id [";"]
        | default [";"]
        | forward [";"]
        ).

reffunDef :=
    reffun id "(" [valueParams] ")" ":" typeExpr
        ( [inline] cmd
        | alias longId ["(" [paramCasts] ")"] [";"]
        | extern id ["(" [paramModes] ")"] [";"]
        | builtin id [";"]
        | forward [";"]
        ).

valDef :=
    val id ":" typeExpr
        [ alias longId [";"]
        | extern id [";"]
        | builtin id [";"]
        ].
```

varDef :=
    `var` *id* ":" *typeExpr*
        [`alias` *longId* [";"]
        | `extern` *id* [";"]
        | `builtin` *id* [";"]
        ].

initDef := `init` *cmd*.

termDef := `term` *cmd*.

paramCasts := (`repr` | "-") {"," (`repr` | "-")}.

paramModes := ("&" | "-") {"," ("&" | "-")}.

cmd : to be provided by the core language.

A.1.3 Modules

module :=
    `module` *globalId* ":" *specExpr*
        "=" [`let` *restrDef* `in`] *moduleExpr* [";"]

restrDef :=
    restrDefSeq
  | *moduleDef*.

restrDefSeq := "{" {*restrDef*} "}" [";"].

moduleDef : see Functors.

A.1.4 Parameter Declarations

moduleParams := *moduleParam* {"," *moduleParam*}.

moduleParam := *id* ":" *specExpr*.

valueParams := *valueParam* {"," *valueParam*}.

valueParam := *inParam* | *inrefParam* | *inoutParam* | *outParam*.

inParam := [`in`] *id* ":" *typeExpr*.
inrefParam := `inref` *id* ":" *typeExpr*.
inoutParam := `inout` *id* ":" *typeExpr*.
outParam := `out` *id* ":" *typeExpr*.

A.1.5 Local Definitions

The following domain does not occur in the functor language grammar, but is provided for use in the core language:

$$
\begin{aligned}
&\mathit{localDef} := \\
&\qquad \mathit{localValDef} \\
&\quad \mid \ \mathit{localVarDef}.
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{localValDef} := \\
&\qquad \texttt{val}\ \mathit{id}\ \text{“}\texttt{:}\text{”}\ \mathit{typeExpr}\ [\text{“}\texttt{`}\text{”}\ \mathit{id}]\ [\mathit{init}]\ [\text{“}\texttt{;}\text{”}]. \\
&\mathit{localVarDef} := \\
&\qquad \texttt{var}\ \mathit{id}\ \text{“}\texttt{:}\text{”}\ \mathit{typeExpr}\ [\text{“}\texttt{`}\text{”}\ \mathit{id}]\ [\mathit{init}]\ [\text{“}\texttt{;}\text{”}].
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{init} := \\
&\qquad \text{“}\texttt{=}\text{”}\ \mathit{valueExpr} \\
&\quad \mid \ \text{“}\texttt{(}\text{”}\ [\mathit{valueExprs}]\ \text{“}\texttt{)}\text{”}.
\end{aligned}
$$

A.1.6 Commands

The following domain does not occur in the functor language grammar, but is provided for use in the core language:

$$
\begin{aligned}
&\mathit{extCmd} := \\
&\qquad \mathit{constrCall} \\
&\quad \mid \ \mathit{destrCall}.
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{constrCall} := \\
&\qquad \texttt{constr}\ \mathit{valueExpr}\ \text{“}\texttt{:}\text{”}\ \mathit{typeExpr}\ [\text{“}\texttt{`}\text{”}\ \mathit{id}]\ [\mathit{init}]\ [\text{“}\texttt{;}\text{”}]. \\
&\mathit{destrCall} := \\
&\qquad \texttt{destr}\ \mathit{valueExpr}\ \text{“}\texttt{:}\text{”}\ \mathit{typeExpr}\ [\text{“}\texttt{`}\text{”}\ \mathit{id}]\ [\text{“}\texttt{(}\text{”}\ [\mathit{valueExprs}]\ \text{“}\texttt{)}\text{”}]\ [\text{“}\texttt{;}\text{”}].
\end{aligned}
$$

init : see Local Definitions.

A.1.7 Expressions

$$\mathit{specExpr} := \mathit{globalId}\ [\text{“}\texttt{[}\text{”}\ \mathit{moduleExprs}\ \text{“}\texttt{]}\text{”}].$$

$$\mathit{moduleExprs} := \mathit{moduleExpr}\ \{\text{“}\texttt{,}\text{”}\ \mathit{moduleExpr}\}.$$

$$\mathit{moduleExpr} := \mathit{moduleId} \mid \mathit{functorExpr}\ \text{“}\texttt{[}\text{”}\ [\mathit{moduleExprs}]\ \text{“}\texttt{]}\text{”}.$$

$$\mathit{moduleId} := \begin{cases} \mathit{globalLongId} & \text{in } \mathit{module} \text{ units.} \\ \mathit{longId} & \text{else.} \end{cases}$$

$$\mathit{functorExpr} := \begin{cases} \mathit{globalId} & \text{in } \mathit{module} \text{ units.} \\ \mathit{id} & \text{else.} \end{cases}$$

$typeExpr$: to be provided by the core language.

$valueExprs := valueExpr \; \{\text{“}\texttt{,}\text{”} \; valueExpr\}.$

$valueExpr$: to be provided by the core language.

The following domain does not occur in the functor language grammar, but is provided for use in the core language grammar for $valueExpr$:

$$extPrefixExpr := \begin{cases} axiomExpr & \text{in } \textit{specification} \text{ units.} \\ reprExpr & \text{in } \textit{functor} \text{ units.} \end{cases}$$

$axiomExpr := oldExpr \mid blockExpr \mid forallExpr \mid existsExpr.$

$oldExpr := \texttt{old} \; prefixExpr.$
$blockExpr := \texttt{let} \; spec \; \texttt{in} \; prefixExpr.$
$forallExpr := \texttt{forall} \; \text{“(”} \; quantifications \; \text{“)”} \; prefixExpr.$
$existsExpr := \texttt{exists} \; \text{“(”} \; quantifications \; \text{“)”} \; prefixExpr.$

$reprExpr := \texttt{repr} \; prefixExpr.$

$quantifications := quantification \; \{\text{“}\texttt{,}\text{”} \; quantification\}.$

$quantification := id \; \text{“}\texttt{:}\text{”} \; typeExpr.$

$prefixExpr$: to be provided by the core language.

A.1.8 Identifiers

$globalLongId := globalId \mid longId.$

This rule is ambiguous: a `unitId` can be reduced to a $globalId$, as well as to an id, and then a $longId$ (see below). However, the syntax tree to be produced by the parser for a `unitId` is always an `id` node, no matter whether the `unitId` occurs as a $globalId$ or as a $longId$. Therefore it does not matter how the above ambiguity is resolved.

$longIds := longId \; \{\text{“}\texttt{,}\text{”} \; longId\}.$

$longId := [moduleExpr \; \text{“}\texttt{::}\text{”}] \; id.$

$globalId := [packageExpr \; \text{“}\texttt{<>}\text{”}] \; \texttt{unitId}.$

$packageExpr := globalId.$

$ids := id \; \{\text{“}\texttt{,}\text{”} \; id\}.$

$id := \texttt{unitId} \mid \texttt{coreId}.$

A.2 Core Language

A.2.1 Commands

cmd :=
  $cmdBlock$
  | $ifCmd$
  | $switchCmd$
  | $whileCmd$
  | $doCmd$
  | $breakCmd$
  | $continueCmd$
  | $returnCmd$
  | $callCmd$
  | $extCmd$.

$cmdBlock$:= "`{`" {$localDef$ | cmd} "`}`" ["`;`"].
$ifCmd$:= `if` "(" $valueExpr$ ")" cmd [`else` cmd].
$switchCmd$:= `switch` "(" $valueExpr$ ")" opt.
$whileCmd$:= `while` "(" $valueExpr$ ")" cmd.
$doCmd$:= `do` cmd `while` "(" $valueExpr$ ")" ["`;`"].
$breakCmd$:= `break` ["`;`"].
$continueCmd$:= `continue` ["`;`"].
$returnCmd$:= `return` [$valueExpr$] ["`;`"].
$callCmd$:= $valueExpr$ ["`;`"].
$extCmd$: provided by the functor language.

$localDef$: provided by the functor language.

opt :=
  $optSeq$
  | $caseOpt$.

$optSeq$:= "`{`" {opt} "`}`" ["`;`"].
$caseOpt$:= $caseDescr$ cmd.

$caseDescr$:=
  `case` $valueExpr$ "`:`" {`case` $valueExpr$ "`:`"}
  | `default` "`:`".

The ambiguity introduced by the rule for $ifCmd$ (the "dangling else") is resolved in the usual way: the `else` is associated with the most recent (`else`-less) `if`.

The ambiguities introduced by the rules for $returnCmd$ and $callCmd$ ("`return f(e)`" might be a either single $returnCmd$, or a sequence of a $returnCmd$ and a $callCmd$; similarly, "`p(e) *c = d`" might be one or two $callCmd$s) is resolved by always using the longest possible expression as a single command or return expression, and always considering an expression occurring directly behind a `return` keyword as part of the $returnCmd$.

A.2.2 Expressions

typeExpr :=
  longId
  | *recordType*
  | *unionType*.

recordType := `Record` "`[`" {*component*} "`]`".

unionType := `Union` "`[`" {*component*} "`]`".

component := *id* "`:`" *typeExpr* ["`,`"].

valueExpr :=
  assignOpExpr
  | *condExpr*
  | *infixExpr*.

assignOpExpr := *infixExpr* *longAssignOp* *valueExpr*.

condExpr := *infixExpr* "`?`" *valueExpr* "`:`" *valueExpr*.

longAssignOp := *assignOp* ["`[`" *moduleExpr* "`]`"].

assignOp := "`=`".

infixExpr :=
  implExpr
  | *disjExpr*
  | *conjExpr*
  | *infixOpExpr*
  | *prefixExpr*.

implExpr := *infixExpr* "`=>`" *infixExpr*.

disjExpr := *infixExpr* "`||`" *infixExpr*.

conjExpr := *infixExpr* "`&&`" *infixExpr*.

infixOpExpr := *infixExpr* *longInfixOp* *infixExpr*.

longInfixOp := *infixOp* ["`[`" *moduleExpr* "`]`"].

infixOp :=
  "`|`"
  | "`^`"
  | "`&`"
  | "`==`" | "`!=`"
  | "`<=`" | "`>=`" | "`<`" | "`>`"
  | "`<<`" | "`>>`"
  | "`+`" | "`-`"
  | "`*`" | "`/`" | "`%`".

The ambiguity introduced by the rule for *infixExpr* is resolved according to given precedences of the infix operators and the fact that '=>' is right associative and all others are left associative. The precedences of the infix operators are indicated by the arrangement of the above rules: every line break signifies an increase of precedence.

prefixExpr :=
  extPrefixExpr
  | *derefExpr*
  | *prefixOpExpr*
  | *postfixExpr*.

extPrefixExpr : provided by the functor language.

derefExpr := *typeExpr* "`@`" *prefixExpr*.

prefixOpExpr := *longPrefixOp* *prefixExpr*.

longPrefixOp := *prefixOp* ["`[`" *moduleExpr* "`]`"].

prefixOp := "`!`" | "`~`" | "`+`" | "`-`" | "`*`" | "`&`".

postfixExpr :=
  applExpr
  | *postfixOpExpr*
  | *primaryExpr*.

applExpr := *postfixExpr* "(" [*valueExprs*] ")".

postfixOpExpr := *postfixExpr* *longPostfixOp*.

valueExprs : see Functor Language Expressions.

longPostfixOp :=
  "`[`" ["`[`" *moduleExpr* "`]`"] *valueExpr* "`]`"
  | "`.`" ["`[`" *moduleExpr* "`]`"] *id*
  | "`->`" ["`[`" *moduleExpr* "`]`"] *id*.

id : provided by the functor language.

primaryExpr :=
  "(" *valueExpr* ")"
  | **`sizeof`** *typeExpr*
  | **`alignof`** *typeExpr*
  | *longLit*
  | *longId*.

longLit := [*moduleExpr* "`::`"] **`lit`**.

longId : provided by the functor language.

moduleExpr : provided by the functor language.

A.3 Comments

The rules for comments in the core language are adapted to the ones chosen for the functor language, so that a consistent combined presentation of the rules, as given below, is possible.

A.3.1 Syntax

From the two terminal classes `scmt` (*separate comment*) and `tcmt` (*trailing comment*) defined in Subsection B.1.2, three kinds of comments may be built up for use in Mona sources:

- A comment *head* is a sequence of separate comments.
- A comment *tail* is a singe trailing comment.
- A comment *end* is a single separate comment.

A.3.2 Placement

Comment *heads* are allowed in Mona sources before

- Mona units (at the beginning of a Mona source file),
- specifications and definitions (also in `let` blocks in `module` units),
- commands,
- local definitions in command blocks, and
- cases in `switch` commands.

A comment *tail* is allowed for all syntax constructs that can have a comment head (except whole units, which get comment *ends*), as well as for

- `input` and `output` axioms (in constructor, destructor, procedure, function, pseudo-function, constant and variable specifications and axioms),
- *`classifier`*`(`*`constructor`*`(`*`selector`*`,...))` items (in `constructed` type axioms),
- `let` blocks in `module` units,
- functor applications,
- component declarations in `Record` or `Union` type expressions, and
- conditional, infix, and quantified value expressions, as well as value expression `let` blocks, array and record indexing expressions, and function applications.

Comment tails have to be placed behind the construct they belong to, with the following exceptions:

- A comment tail for any braced sequence (specification or definition sequence—also in `module` units, statement block or case sequence) comes directly after the opening brace '`{`'.
- A comment tail for any `let` block (specification, definition, or value expression block, `let` block in `module` units) comes directly after the keyword `in`.
- A comment tail for any constructor, destructor, procedure, function or value specification or axiom comes before any `input` or `output` axiom.
- A comment tail for any `constructed` type axiom comes before the list of *`classifier`*`(`*`constructor`*`(`*`selector`*`,...))` items.
- A comment tail for any definition with a core procedure or function body (constructor, destructor, procedure or function, as well as module initializer or terminator definition) comes before this body (before the keyword `inline`, if it is there).
- A comment tail for an `if`, `switch` or `while` statement comes directly after the parenthesized expression.
- A comment tail for a case in a `switch` statement comes before the command to execute in the described case (i.e., after the colon '`:`' behind the `case` description sequence or the `default`).
- A comment tail for a conditional value expression comes directly after the question mark '`?`'.
- A comment tail for a quantified value expression comes directly after the quantifier (or, more precisely, after the parenthesized list of variables bound by the quantifier).
- A comment tail for any infix value expression comes directly after the infix operator.

A comment *end*, finally, is allowed at the very end of a Mona unit (i.e., at the end of any Mona source file).

Appendix B

Lexical Syntax

In this appendix specifications of the lexical syntax of both functor and core language, as used for scanning these languages, are given in the form of regular definitions using the meta-symbols

- '␣' for a space character,
- '$\mapsto$' for a tab character,
- '$\hookleftarrow$' for a newline character,
- 're^*' for zero or more occurrences of anything matching re,
- '$re^?$' for zero or one occurrence of anything matching re,
- 're^+' for one or more occurrences of anything matching re,
- '$re_1 \mid re_2$' for anything that matches re_1 or re_2,
- '(' and ')' for grouping, and
- '$:=$' to separate the left and right hand side of a definition,

where '*', '$^?$' and '$^+$' have higher precedence than concatenation, which in turn has higher precedence than '|'.

B.1 Functor Language

The input for the Mona parser is divided into tokens by collecting sequences of characters matching either one of the keywords or other terminal symbols occurring in the grammar, or one of the right hand sides of the regular definitions given in the following subsections, which describe the terminal classes of the functor language grammar. Ambiguities are resolved by choosing the longest possible match as next token, i.e., the input character sequence '`axiomtype`', e.g., is scanned as *`unitId`*, and not as sequence of the two keywords '`axiom`' and '`type`'. White space, defined as

$$whiteSpace := (\text{␣} \mid \mapsto \mid \hookleftarrow)^*$$

(a sequence of space, tab or newline characters), must be used to separate tokens that would collapse to a single one without separation (as, e.g., the '`axiom`' and '`type`' above), and is otherwise ignored.

B.1.1 Unit Identifiers

$$\texttt{unitId} := letter\ (letter \mid dec)^*$$

$$\begin{aligned} letter := \\ & \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \mid \texttt{G} \mid \texttt{H} \mid \texttt{I} \mid \texttt{J} \mid \texttt{K} \mid \texttt{L} \mid \texttt{M} \\ \mid & \texttt{N} \mid \texttt{O} \mid \texttt{P} \mid \texttt{Q} \mid \texttt{R} \mid \texttt{S} \mid \texttt{T} \mid \texttt{U} \mid \texttt{V} \mid \texttt{W} \mid \texttt{X} \mid \texttt{Y} \mid \texttt{Z} \\ \mid & \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{g} \mid \texttt{h} \mid \texttt{i} \mid \texttt{j} \mid \texttt{k} \mid \texttt{l} \mid \texttt{m} \\ \mid & \texttt{n} \mid \texttt{o} \mid \texttt{p} \mid \texttt{q} \mid \texttt{r} \mid \texttt{s} \mid \texttt{t} \mid \texttt{u} \mid \texttt{v} \mid \texttt{w} \mid \texttt{x} \mid \texttt{y} \mid \texttt{z} \end{aligned}$$

$$dec := 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

B.1.2 Comments

The following terminal classes are referred to in Section A.3.

$$\texttt{tcmt} := cmtLine^+$$

$$\texttt{scmt} := \hookleftarrow\ cmtLine^+$$

$$cmtLine := blanks\ \texttt{//.}^*\ \hookleftarrow$$

$$blanks := (\text{␣} \mid \mapsto)^*$$

(The definition of $\texttt{scmt}$ is not strictly correct, since also a sequence of $cmtLine$s at the beginning of a file—without any newline characters before—is scanned as a separate comment.)

B.2 Core Language

In addition to the keywords and other terminal symbols occurring in the core language grammar, the core language provides an extended identifier terminal class, as well as a terminal class for number and string literals.

B.2.1 Core Identifiers

$\texttt{coreId} := \mathit{coreName} \mid \mathit{opName} \mid \mathit{chrName} \mid \mathit{digName} \mid \mathit{emptyName}$

$\mathit{coreName} := \texttt{unitId}\ \texttt{\_}\ (\mathit{letter} \mid \texttt{\_} \mid \mathit{dec})^*$

$\mathit{opName} := \texttt{operator}\ \text{␣}\ \mathit{op}$

$\mathit{op} :=$
$\quad \texttt{=}$
$\mid\ \texttt{|} \mid \texttt{^} \mid \texttt{\&}$
$\mid\ \texttt{==} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} \mid \texttt{<} \mid \texttt{>}$
$\mid\ \texttt{<<} \mid \texttt{>>}$
$\mid\ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%}$
$\mid\ \texttt{!} \mid \texttt{~} \mid \texttt{[]}$
$\mid\ \texttt{.}\mathit{id} \mid \texttt{->}\mathit{id}$
$\mid\ \texttt{0d} \mid \texttt{0D}$
$\mid\ \texttt{0x} \mid \texttt{0X}$
$\mid\ \texttt{0f} \mid \texttt{0I} \mid \texttt{0F} \mid \texttt{1E} \mid \texttt{E-}$
$\mid\ \texttt{"}$

$\mathit{id} := \texttt{unitId} \mid \texttt{coreId}$

$\texttt{unitId}$: functor language terminal class

$\mathit{chrName} := \texttt{'}\mathit{chr}\texttt{'}$

$\mathit{digName} := \texttt{\#}\ \mathit{hex}$

$\mathit{emptyName} := \texttt{''}$

B.2.2 Literals

$\texttt{lit} := \mathit{decNum} \mid \mathit{hexNum} \mid \mathit{fltNum} \mid \mathit{chrStr}$

$\mathit{decNum} := (\texttt{0d})^?\ \mathit{dec}^+$

$\mathit{hexNum} := \texttt{0x}\ \mathit{hex}^+$

$\mathit{fltNum} := \texttt{0f}\ \mathit{int}\ \mathit{frc}^?\ \mathit{exp}^?$
$\mathit{int} := \mathit{dec}^+$
$\mathit{frc} := \texttt{.}\ \mathit{dec}^*$
$\mathit{exp} := \texttt{e}\ (\texttt{+}^? \mid \texttt{-})\ \mathit{dec}^+$

$\mathit{chrStr} := \texttt{"}\mathit{chr}^*\texttt{"}$

B.2.3 Characters and Digits

```
chr :=
    \0 | \t | \n | ␣
  | ! | \" | # | $ | % | & | \'
  | ( | ) | * | + | , | - | . | /
  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  | : | ; | < | = | > | ? | @
  | A | B | C | D | E | F | G | H | I | J | K | L | M
  | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
  | [ | \\ | ] | ^ | _ | `
  | a | b | c | d | e | f | g | h | i | j | k | l | m
  | n | o | p | q | r | s | t | u | v | w | x | y | z
  | { | | | } | ~
```

```
dec := 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
hex := dec | A | B | C | D | E | F
```

Appendix C

Abstract Syntax

This appendix contains a complete description of the syntax trees produced by the Mona parser. The possible trees are specified in form of a grammar (using the notation introduced in Appendix A), where each rule

$$label := subtree_1 \cdots subtree_n.$$

with a concatenation of items $subtree_1, \ldots, subtree_n$ as right hand side specifies a tree with root label $label$ and subtrees as specified by $subtree_1, \ldots, subtree_n$. A rule with the empty string εas right hand side specifies a tree node without subtrees. In other words: the possible syntax trees produced by the Mona parser are the possible derivation trees according to the grammar given below. However, the following two exceptions apply:

- Uppercase nonterminal symbols in the grammar du not correspond to syntax tree nodes, but are used as 'macros', which have to be 'expanded' where they occur. A rule like

  $$n := o\ M\ p.$$

  with M being defined as

  $$M := a \mid b \mid c.$$

  is thus to be interpreted as

  $$n := o\ (a \mid b \mid c)\ p.$$

- An optional item "[n]" occurring in the grammar represents a tree with either an *opt1* root label and n as subtree, or a node with label *opt0* and no subtrees. One could say that "[n]" represents "(*opt1* | *opt0*)", with *opt1* being defined as

  $$opt1 := n.$$

  in this context, and *opt0* being

  $$opt0 := \varepsilon.$$

However, this can not be formulated in a context-free grammar, since the definition of $opt1$ depends on its context.

Macros were introduced to keep a correspondence with the concrete syntax grammar (Appendix A), while avoiding unnecessary nodes in the syntax tree for grammar rules with '|' as top operator on the right hand rule side.

Using the $opt0$ and $opt1$ tree node labels avoids having to introduce separate top node labels for optional occurrences of subtrees, while still achieving a fixed number of subtrees (in contrast to just letting the optional item occur as subtree or not), which simplifies processing of the tree.

Leaf nodes of the syntax tree are either nodes whose labels are defined by rules with empty right hand side ε, like, e.g., $breakCmd$, or one of the following:

- an identifier node `id` containing a reference to the identifier table entry that gives the textual representation of the identifier,
- a comment node (either `lcmt` for a *leading* comment, or `tcmt` for a *trailing* comment) containing a reference to the comment table entry that gives the comment text, or
- a line number node `lineNr` containing an integer source line number.

Comment and line number nodes are not mentioned in the following grammar, but are treated separately in Section C.3.

The root node of the syntax tree generated by the Mona parser is always a $UNIT$ node, i.e., a node with label $specification$, $functor$ or $module$ (cf. below).

C.1 Functor Language

$$
\begin{aligned}
&UNIT := \\
&\quad specification \\
&\quad |\ functor \\
&\quad |\ module.
\end{aligned}
$$

C.1.1 Specifications

$$specification := GLOBAL\_ID\ moduleParams\ SPEC.$$

$SPEC :=$
$\quad specSeq$
$\quad |\ specBlock$
$\quad |\ include\ \ |\ \ includeOnly\ \ |\ \ includeExcept$
$\quad |\ functorUse$
$\quad |\ moduleSpec$
$\quad |\ import\ \ |\ \ importOnly\ \ |\ \ importExcept$
$\quad |\ typeSpec$
$\quad |\ constrSpec$
$\quad |\ destrSpec$
$\quad |\ procSpec$
$\quad |\ funSpec$
$\quad |\ reffunSpec$
$\quad |\ valSpec$
$\quad |\ varSpec$
$\quad |\ moduleAxiom$
$\quad |\ typeAxiom\ \ |\ \ equalityAxiom\ \ |\ \ truthAxiom\ \ |\ \ caseAxiom\ \ |$
$\qquad addressAxiom\ \ |\ \ truevalAxiom\ \ |\ \ constructedAxiom$
$\quad |\ constrAxiom$
$\quad |\ destrAxiom$
$\quad |\ procAxiom$
$\quad |\ funAxiom$
$\quad |\ reffunAxiom$
$\quad |\ valAxiom$
$\quad |\ varAxiom$
$\quad |\ generalAxiom.$

$specSeq := \{SPEC\}.$

$specBlock := SPEC\ SPEC.$

$include := SPEC\_EXPR.$

$includeOnly := SPEC\_EXPR\ ids.$

$includeExcept := SPEC\_EXPR\ ids.$

$functorUse$: see Functors.

$moduleSpec :=$ `id` $SPEC\_EXPR\ [MODULE\_EXPR].$

$import$: see Functors.

$importOnly$: see Functors.

$importExcept$: see Functors.

$typeSpec :=$ `id` $[TYPE\_EXPR \mid notEqType \mid builtin].$

$constrSpec :=$
`id` $TYPE\_EXPR$ [`id`] $valueParams\ [inputAxiom]\ [outputAxiom].$

$destrSpec :=$
`id` $TYPE\_EXPR$ [`id`] $valueParams\ [inputAxiom]\ [outputAxiom].$

$procSpec$:= `id` $valueParams$ [$inputAxiom$] [$outputAxiom$].

$funSpec$:=
    `id` $valueParams$ [`id`] $TYPE\_EXPR$ [$VALUE\_EXPR$]
        [$inputAxiom$] [$outputAxiom$].
$reffunSpec$:=
    `id` $valueParams$ [`id`] $TYPE\_EXPR$ [$VALUE\_EXPR$]
        [$inputAxiom$] [$outputAxiom$].

$valSpec$:= `id` $TYPE\_EXPR$ [$VALUE\_EXPR$] [$outputAxiom$].
$varSpec$:= `id` $TYPE\_EXPR$ [$VALUE\_EXPR$] [$outputAxiom$].

$moduleAxiom$:= $MODULE\_EXPR$ $MODULE\_EXPR$.

$typeAxiom$:= $TYPE\_EXPR$ ($TYPE\_EXPR$ | $notEqType$).
$equalityAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$truthAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$caseAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$addressAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$truevalAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$falsevalAxiom$:= $TYPE\_EXPR$ $LONG\_ID$.
$constructedAxiom$:= $TYPE\_EXPR$ $classifiedConstrs$ [$defaultConstr$].

$constrAxiom$:=
    `id` $TYPE\_EXPR$ [`id`] $valueParams$ [$inputAxiom$] [$outputAxiom$].
$destrAxiom$:=
    `id` $TYPE\_EXPR$ [`id`] $valueParams$ [$inputAxiom$] [$outputAxiom$].

$procAxiom$:= $LONG\_ID$ $valueParams$ [$inputAxiom$] [$outputAxiom$].

$funAxiom$:=
    $LONG\_ID$ $valueParams$ [`id`] $TYPE\_EXPR$ [$VALUE\_EXPR$]
        [$inputAxiom$] [$outputAxiom$].
$reffunAxiom$:=
    $LONG\_ID$ $valueParams$ [`id`] $TYPE\_EXPR$ [$VALUE\_EXPR$]
        [$inputAxiom$] [$outputAxiom$].

$valAxiom$:= $LONG\_ID$ $TYPE\_EXPR$ [$VALUE\_EXPR$] [$outputAxiom$].
$varAxiom$:= $LONG\_ID$ $TYPE\_EXPR$ [$VALUE\_EXPR$] [$outputAxiom$].

$generalAxiom$:= $VALUE\_EXPR$.

$notEqType$:= $TYPE\_EXPR$.
$builtin$: see Functors.

$inputAxiom := VALUE\_EXPR.$

$outputAxiom := VALUE\_EXPR.$

$classifiedConstrs := \{classifiedConstr\}.$

$classifiedConstr := LONG\_ID\ CONSTR\ longIds.$

$defaultConstr := CONSTR\ longIds.$

$CONSTR := LONG\_ID \mid restrConstr.$

$restrConstr := LONG\_ID\ LONG\_ID.$

C.1.2 Functors

$functor := GLOBAL\_ID\ moduleParams\ SPEC\_EXPR\ DEF.$

$$
\begin{aligned}
DEF :=& \\
&\quad defSeq \\
&\mid defBlock \\
&\mid functorUse \\
&\mid moduleDef \\
&\mid import \mid importOnly \mid importExcept \\
&\mid typeDef \\
&\mid constrDef \\
&\mid destrDef \\
&\mid procDef \\
&\mid funDef \\
&\mid reffunDef \\
&\mid valDef \\
&\mid varDef \\
&\mid initDef \\
&\mid termDef.
\end{aligned}
$$

$defSeq := \{DEF\}.$

$defBlock := DEF\ DEF.$

$functorUse := \texttt{id}\ moduleParams\ SPEC\_EXPR\ GLOBAL\_ID.$

$moduleDef := \texttt{id}\ SPEC\_EXPR\ (MODULE\_EXPR \mid forward).$

$import := MODULE\_EXPR.$

$importOnly := MODULE\_EXPR\ ids.$

$importExcept := MODULE\_EXPR\ ids.$

$typeDef := \texttt{id}\ (TYPE\_EXPR \mid newType \mid builtin \mid forward).$

$$
\begin{aligned}
&\mathit{constrDef} :=\\
&\quad \texttt{id}\ \mathit{TYPE\_EXPR}\ [\texttt{id}]\ \mathit{valueParams}\\
&\qquad (\ \mathit{CMD}\ \mid\ \mathit{inlineProc}\\
&\qquad \mid \mathit{constrAlias}\\
&\qquad \mid \mathit{forward}\\
&\qquad).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{destrDef} :=\\
&\quad \texttt{id}\ \mathit{TYPE\_EXPR}\ [\texttt{id}]\ \mathit{valueParams}\\
&\qquad (\ \mathit{CMD}\ \mid\ \mathit{inlineProc}\\
&\qquad \mid \mathit{destrAlias}\\
&\qquad \mid \mathit{forward}\\
&\qquad).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{procDef} :=\\
&\quad \texttt{id}\ \mathit{valueParams}\\
&\qquad (\ \mathit{CMD}\ \mid\ \mathit{inlineProc}\\
&\qquad \mid \mathit{procAlias}\\
&\qquad \mid \mathit{externProc}\\
&\qquad \mid \mathit{builtin}\\
&\qquad \mid \mathit{forward}\\
&\qquad).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{funDef} :=\\
&\quad \texttt{id}\ \mathit{valueParams}\ [\texttt{id}]\ \mathit{TYPE\_EXPR}\\
&\qquad (\ \mathit{CMD}\ \mid\ \mathit{inlineFun}\\
&\qquad \mid \mathit{funAlias}\\
&\qquad \mid \mathit{externFun}\\
&\qquad \mid \mathit{builtin}\\
&\qquad \mid \mathit{default}\\
&\qquad \mid \mathit{forward}\\
&\qquad).
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{reffunDef} :=\\
&\quad \texttt{id}\ \mathit{valueParams}\ [\varepsilon]\ \mathit{TYPE\_EXPR}\\
&\qquad (\ \mathit{CMD}\ \mid\ \mathit{inlineFun}\\
&\qquad \mid \mathit{funAlias}\\
&\qquad \mid \mathit{externFun}\\
&\qquad \mid \mathit{builtin}\\
&\qquad \mid \mathit{forward}\\
&\qquad).
\end{aligned}
$$

(The '$[\varepsilon]$' above was once an '[`id`]', but an `id` is no longer accepted at this place—the parser just generates the *opt0* node indicating that no `id` is present.)

$$\mathit{valDef} := \texttt{id}\ \mathit{TYPE\_EXPR}\ [\mathit{valAlias} \mid \mathit{externVal} \mid \mathit{builtin}].$$

$$\mathit{varDef} := \texttt{id}\ \mathit{TYPE\_EXPR}\ [\mathit{valAlias} \mid \mathit{externVal} \mid \mathit{builtin}].$$

$$\mathit{initDef} := \mathit{CMD}.$$

$$\mathit{termDef} := \mathit{CMD}.$$

$newType := TYPE\_EXPR.$

$inlineProc := CMD.$
$inlineFun := CMD.$

$constrAlias := TYPE\_EXPR\ [\texttt{id}]\ [paramCasts].$
$destrAlias := TYPE\_EXPR\ [\texttt{id}]\ [paramCasts].$
$procAlias := LONG\_ID\ [paramCasts].$
$funAlias := LONG\_ID\ [paramCasts].$
$valAlias := LONG\_ID.$

$externProc := \texttt{id}\ [paramModes].$
$externFun := \texttt{id}\ [paramModes].$
$externVal := \texttt{id}.$

$builtin := \texttt{id}.$

$default := \varepsilon.$

$forward := \varepsilon.$

$paramCasts := \{trueFlag \mid falseFlag\}.$

$paramModes := \{trueFlag \mid falseFlag\}.$

$trueFlag := \varepsilon.$
$falseFlag := \varepsilon.$

CMD : to be defined by the core language.

C.1.3 Modules

$module :=$
$\quad GLOBAL\_ID\ SPEC\_EXPR\ (MODULE\_EXPR \mid moduleBlock).$

$moduleBlock := RESTR\_DEF\ \ MODULE\_EXPR.$

$RESTR\_DEF :=$
$\quad restrDefSeq$
$\mid moduleDef.$

$restrDefSeq := \{RESTR\_DEF\}.$

$moduleDef$: see Functors.

C.1.4 Parameter Declarations

$moduleParams := \{moduleParam\}.$

$moduleParam := \texttt{id}\ SPEC\_EXPR.$

$valueParams := \{VALUE\_PARAM\}.$

$VALUE\_PARAM := inParam \mid inrefParam \mid inoutParam \mid outParam.$

$inParam := \texttt{id}\ TYPE\_EXPR.$
$inrefParam := \texttt{id}\ TYPE\_EXPR.$
$inoutParam := \texttt{id}\ TYPE\_EXPR.$
$outParam := \texttt{id}\ TYPE\_EXPR.$

C.1.5 Local Definitions

The following domain is provided for use in the core language:

$$LOCAL\_DEF ::= localValDef \mid localVarDef.$$

$localValDef ::=$ `id` $TYPE\_EXPR$ [`id`] $INIT.$

$localVarDef ::=$ `id` $TYPE\_EXPR$ [`id`] $INIT.$

$INIT ::= VALUE\_EXPR \mid valueExprs.$

C.1.6 Commands

The following domain is provided for use in the core language:

$$EXT\_CMD ::= constrCall \mid destrCall.$$

$constrCall ::= VALUE\_EXPR\ TYPE\_EXPR$ [`id`] $INIT.$

$destrCall ::= VALUE\_EXPR\ TYPE\_EXPR$ [`id`] $valueExprs.$

$INIT$: see Local Definitions.

C.1.7 Expressions

$SPEC\_EXPR ::= GLOBAL\_ID \mid specAppl.$

$specAppl ::= GLOBAL\_ID\ moduleExprs.$

$moduleExprs ::= \{MODULE\_EXPR\}.$

$MODULE\_EXPR ::= MODULE\_ID \mid functorAppl.$

$$MODULE\_ID ::= \begin{cases} GLOBAL\_LONG\_ID & \text{in } module \text{ units.} \\ LONG\_ID & \text{else.} \end{cases}$$

$functorAppl ::= FUNCTOR\_EXPR\ moduleExprs.$

$$FUNCTOR\_EXPR ::= \begin{cases} GLOBAL\_ID & \text{in } module \text{ units.} \\ \texttt{id} & \text{else.} \end{cases}$$

$TYPE\_EXPR$: to be defined by the core language.

$valueExprs := \{VALUE\_EXPR\}.$

$VALUE\_EXPR$: to be defined by the core language.

The following domain is provided for use in the core language $VALUE\_EXPR$ domain:

$$EXT\_VALUE\_EXPR := \begin{cases} AXIOM\_EXPR & \text{in } \textit{specification} \text{ units.} \\ reprExpr & \text{in } \textit{functor} \text{ units.} \end{cases}$$

$AXIOM\_EXPR := oldExpr \mid blockExpr \mid forallExpr \mid existsExpr.$

$oldExpr := VALUE\_EXPR.$
$blockExpr := SPEC\ \ VALUE\_EXPR.$
$forallExpr := quantifications\ \ VALUE\_EXPR.$
$existsExpr := quantifications\ \ VALUE\_EXPR.$

$reprExpr := VALUE\_EXPR.$

$quantifications := quantification\ \{quantification\}.$

$quantification := \texttt{id}\ TYPE\_EXPR.$

C.1.8 Identifiers

$GLOBAL\_LONG\_ID := GLOBAL\_ID \mid moduleQualId.$

$longIds := \{LONG\_ID\}.$

$LONG\_ID := \texttt{id} \mid moduleQualId.$

$moduleQualId := MODULE\_EXPR\ \texttt{id}.$

$GLOBAL\_ID := \texttt{id} \mid packageQualId.$

$packageQualId := PACKAGE\_EXPR\ \texttt{id}.$

$PACKAGE\_EXPR := GLOBAL\_ID.$

$ids := \texttt{id}\ \{\texttt{id}\}.$

C.2 Core Language

C.2.1 Commands

$$\begin{aligned} CMD :=\ & \\ & cmdBlock \\ \mid\ & ifCmd \mid ifElseCmd \\ \mid\ & switchCmd \\ \mid\ & whileCmd \\ \mid\ & doCmd \\ \mid\ & breakCmd \\ \mid\ & continueCmd \\ \mid\ & returnCmd \\ \mid\ & callCmd \\ \mid\ & EXT\_CMD. \end{aligned}$$

$cmdBlock := \{LOCAL\_DEF \mid CMD\}.$

$ifCmd := VALUE\_EXPR\ CMD.$

$ifElseCmd := VALUE\_EXPR\ CMD\ CMD.$

$switchCmd := VALUE\_EXPR\ OPT.$

$whileCmd := VALUE\_EXPR\ CMD.$

$doCmd := CMD\ VALUE\_EXPR.$

$breakCmd := \varepsilon.$

$continueCmd := \varepsilon.$

$returnCmd := [VALUE\_EXPR].$

$callCmd := VALUE\_EXPR.$

EXT_CMD : defined by the functor language.

$LOCAL\_DEF$: defined by the functor language.

$$
\begin{aligned}
OPT := &\\
&optSeq\\
\mid\ &caseOpt.
\end{aligned}
$$

$optSeq := \{OPT\}.$

$caseOpt := caseDescr\ CMD.$

$caseDescr := \{VALUE\_EXPR\}.$

C.2.2 Expressions

$$
\begin{aligned}
TYPE\_EXPR := &\\
&LONG\_ID\\
\mid\ &recordType\\
\mid\ &unionType.
\end{aligned}
$$

$recordType := \{component\}.$

$unionType := \{component\}.$

$component :=$ `id` $TYPE\_EXPR.$

$$
\begin{aligned}
VALUE\_EXPR := &\\
&condExpr\\
\mid\ &implExpr\\
\mid\ &disjExpr\\
\mid\ &conjExpr\\
\mid\ &derefExpr\\
\mid\ &applExpr\\
\mid\ &sizeExpr\\
\mid\ &alignExpr\\
\mid\ &EXT\_VALUE\_EXPR\\
\mid\ &LONG\_ID.
\end{aligned}
$$

$condExpr$:= $VALUE\_EXPR$ $VALUE\_EXPR$ $VALUE\_EXPR$.

$implExpr$:= $VALUE\_EXPR$ $VALUE\_EXPR$.

$disjExpr$:= $VALUE\_EXPR$ $VALUE\_EXPR$.

$conjExpr$:= $VALUE\_EXPR$ $VALUE\_EXPR$.

$derefExpr$:= $TYPE\_EXPR$ $VALUE\_EXPR$.

$applExpr$:= $VALUE\_EXPR$ $valueExprs$.

$sizeExpr$:= $TYPE\_EXPR$.

$alignExpr$:= $TYPE\_EXPR$.

EXT_VALUE_EXPR : defined by the functor language.

$LONG\_ID$: defined by the functor language.

$valueExprs$:= see Functor Language Expressions.

C.3 Comments and Line Numbers

Comments in the source files are included into the syntax tree by the parser as either `lcmt` nodes (produced for separate comments used as comment *head* or `tcmt` nodes (for comment *tails*)—see Section A.3 for a description of the allowed concrete syntax of comments in Mona source files. Comment nodes get attached to the end of the subtree sequence of the construct to which the comment belongs. This results in the following rules.

A sequence of `lcmt` nodes may be attached to the son sequences of all nodes of the classes $UNIT$, $SPEC$, DEF, $RESTR\_DEF$, CMD and OPT.

A `tcmt` may occur at any position where `lcmt` nodes are allowed (behind all `lcmt` nodes, if there are some), and additionally be attached to the son sequences of

- $moduleBlock$ nodes,
- the EXT_VALUE_EXPR nodes $blockExpr$, $forallExpr$ and $existsExpr$,
- $classifiedConstr$, $defaultConstr$, $inputAxiom$, $outputAxiom$ and $component$ nodes, as well as
- the $VALUE\_EXPR$ nodes $condExpr$, $implExpr$, $disjExpr$, $conjExpr$, $applExpr$, $sizeExpr$ and $alignExpr$.

The parser includes source line number information in the form of special `lineNr` nodes in the syntax tree, which can be used by compiler modules processing the tree for letting error messages refer to the corresponding source lines. A `lineNr` node is attached to the son sequences of all nodes which may have `lcmt` nodes (before any comment node, if there is one).

Appendix D

Built-in Types and Operations

In this appendix, all built-in items provided by Mona for use in `builtin` definitions are listed. Their interfaces are given in Mona specification syntax. Also the special axioms (specifying conversion functions used for conditions, etc.) assumed for the different built-in types are listed. Finally, the C++ code produced by the code generator for each of the built-in items is given.

D.1 Booleans

```
type Bool    ⟶    int

val false: Bool    ⟶    0
val true:  Bool    ⟶    1

fun operator ~(b: Bool):          Bool    ⟶    _ = ~b
fun operator &(b: Bool, c: Bool): Bool    ⟶    _ = b & c
fun operator |(b: Bool, c: Bool): Bool    ⟶    _ = b | c
fun operator ^(b: Bool, c: Bool): Bool    ⟶    _ = b ^ c

fun Bool(b: Bool): Bool    ⟶    _ = b

fun operator ==(b: Bool, c: Bool): Bool    ⟶    _ = b == c
fun operator !=(b: Bool, c: Bool): Bool    ⟶    _ = b != c
fun operator < (b: Bool, c: Bool): Bool    ⟶    _ = b <  c
fun operator <=(b: Bool, c: Bool): Bool    ⟶    _ = b <= c
fun operator > (b: Bool, c: Bool): Bool    ⟶    _ = b >  c
fun operator >=(b: Bool, c: Bool): Bool    ⟶    _ = b >= c

constr b: Bool()          ⟶    (void)0
constr b: Bool(c: Bool)   ⟶    b = c

destr  b: Bool()          ⟶    (void)0

axiom type Bool truth    Bool
axiom type Bool trueval  true
axiom type Bool falseval false

proc operator =(out b: Bool, c: Bool)    ⟶    b = c
```

D.2 Characters

```
type Char   ⟶   char

val '\0': Char    ⟶    '\0'
val '\t': Char    ⟶    '\t'
val '\n': Char    ⟶    '\n'

val ' ':  Char    ⟶    ' '
val '!':  Char    ⟶    '!'
val '\"': Char    ⟶    '\"'
val '#':  Char    ⟶    '#'
val '$':  Char    ⟶    '$'
val '%':  Char    ⟶    '%'
val '&':  Char    ⟶    '&'
val '\'': Char    ⟶    '\''
val '(':  Char    ⟶    '('
val ')':  Char    ⟶    ')'
val '*':  Char    ⟶    '*'
val '+':  Char    ⟶    '+'
val ',':  Char    ⟶    ','
val '-':  Char    ⟶    '-'
val '.':  Char    ⟶    '.'
val '/':  Char    ⟶    '/'

val '0':  Char    ⟶    '0'
val '1':  Char    ⟶    '1'
val '2':  Char    ⟶    '2'
val '3':  Char    ⟶    '3'
val '4':  Char    ⟶    '4'
val '5':  Char    ⟶    '5'
val '6':  Char    ⟶    '6'
val '7':  Char    ⟶    '7'
val '8':  Char    ⟶    '8'
val '9':  Char    ⟶    '9'

val ':':  Char    ⟶    ':'
val ';':  Char    ⟶    ';'
val '<':  Char    ⟶    '<'
val '=':  Char    ⟶    '='
val '>':  Char    ⟶    '>'
val '?':  Char    ⟶    '?'
val '@':  Char    ⟶    '@'

val 'A':  Char    ⟶    'A'
val 'B':  Char    ⟶    'B'
val 'C':  Char    ⟶    'C'
val 'D':  Char    ⟶    'D'
val 'E':  Char    ⟶    'E'
val 'F':  Char    ⟶    'F'
val 'G':  Char    ⟶    'G'
```

```
val 'H':  Char   ⟶   'H'
val 'I':  Char   ⟶   'I'
val 'J':  Char   ⟶   'J'
val 'K':  Char   ⟶   'K'
val 'L':  Char   ⟶   'L'
val 'M':  Char   ⟶   'M'
val 'N':  Char   ⟶   'N'
val 'O':  Char   ⟶   'O'
val 'P':  Char   ⟶   'P'
val 'Q':  Char   ⟶   'Q'
val 'R':  Char   ⟶   'R'
val 'S':  Char   ⟶   'S'
val 'T':  Char   ⟶   'T'
val 'U':  Char   ⟶   'U'
val 'V':  Char   ⟶   'V'
val 'W':  Char   ⟶   'W'
val 'X':  Char   ⟶   'X'
val 'Y':  Char   ⟶   'Y'
val 'Z':  Char   ⟶   'Z'

val '[':  Char   ⟶   '['
val '\\': Char   ⟶   '\\'
val ']':  Char   ⟶   ']'
val '^':  Char   ⟶   '^'
val '_':  Char   ⟶   '_'
val '`':  Char   ⟶   '`'

val 'a':  Char   ⟶   'a'
val 'b':  Char   ⟶   'b'
val 'c':  Char   ⟶   'c'
val 'd':  Char   ⟶   'd'
val 'e':  Char   ⟶   'e'
val 'f':  Char   ⟶   'f'
val 'g':  Char   ⟶   'g'
val 'h':  Char   ⟶   'h'
val 'i':  Char   ⟶   'i'
val 'j':  Char   ⟶   'j'
val 'k':  Char   ⟶   'k'
val 'l':  Char   ⟶   'l'
val 'm':  Char   ⟶   'm'
val 'n':  Char   ⟶   'n'
val 'o':  Char   ⟶   'o'
val 'p':  Char   ⟶   'p'
val 'q':  Char   ⟶   'q'
val 'r':  Char   ⟶   'r'
val 's':  Char   ⟶   's'
val 't':  Char   ⟶   't'
val 'u':  Char   ⟶   'u'
val 'v':  Char   ⟶   'v'
val 'w':  Char   ⟶   'w'
val 'x':  Char   ⟶   'x'
```

```
val 'y':  Char    ⟶    'y'
val 'z':  Char    ⟶    'z'

val '{':  Char    ⟶    '{'
val '|':  Char    ⟶    '|'
val '}':  Char    ⟶    '}'
val '~':  Char    ⟶    '~'

fun Char(c: Char): Char    ⟶    _ = c
fun Char(i: Int):  Char    ⟶    _ = (char)i

fun operator ==(c: Char, d: Char): Bool    ⟶    _ = c == d
fun operator !=(c: Char, d: Char): Bool    ⟶    _ = c != d
fun operator < (c: Char, d: Char): Bool    ⟶    _ = c <  d
fun operator <=(c: Char, d: Char): Bool    ⟶    _ = c <= d
fun operator > (c: Char, d: Char): Bool    ⟶    _ = c >  d
fun operator >=(c: Char, d: Char): Bool    ⟶    _ = c >= d

constr c: Char()           ⟶    (void)0
constr c: Char(d: Char)    ⟶    c = d

destr  c: Char()           ⟶    (void)0

axiom type Char case Int

proc operator =(out c: Char, d: Char)    ⟶    c = d
```

D.3 Strings

```
type String    ⟶    char const*

val '': String    ⟶    ""

fun operator "(c: Char,   t: String): String
    ⟶    ({char *_p = (char*)malloc(2 + strlen(t));
           _ = _p;
           *_p = c;
           strcpy(_p + 1, t);
           })
fun operator "(s: String, t: String): String
    ⟶    ({int _l = strlen(s);
           char *_p = (char*)malloc(1 + _l + strlen(t));
           _ = _p;
           strcpy(_p, s);
           strcpy(_p + _l, t);
           })

fun String(s: String): String
    ⟶    ({char *_p = (char*)malloc(1 + strlen(s));
           _ = _p;
           strcpy(_p, s);
           })
```

```
fun String(i: Int):    String
    ⟶      ({int _r = i;
             int _n = _r < 0;
             char *_p;
             int _s = _r;
             int _l = 1;
             while (_s /= 10) _l++;
             *_p = (char*)malloc(1 + _n + _l);
             _ = _p;
             if (_n) _r = -_r;
             _p += _n + _l;
             *_p = '\0';
             do *--_p = '0' + _r%10; while (_r /= 10);
             if (_n) *--_p = '-';
             })
```

```
fun operator[](s: String, i: Int): Char    ⟶    _ = s[i]
```

```
destr s: String()    ⟶    free(s)
```

D.4 Integers

```
type Byte     ⟶    signed char
type Short    ⟶    short
type Int      ⟶    int
type Long     ⟶    long
```

```
val #0: Int    ⟶     0
val #1: Int    ⟶     1
val #2: Int    ⟶     2
val #3: Int    ⟶     3
val #4: Int    ⟶     4
val #5: Int    ⟶     5
val #6: Int    ⟶     6
val #7: Int    ⟶     7
val #8: Int    ⟶     8
val #9: Int    ⟶     9
val #A: Int    ⟶    10
val #B: Int    ⟶    11
val #C: Int    ⟶    12
val #D: Int    ⟶    13
val #E: Int    ⟶    14
val #F: Int    ⟶    15
```

```
fun operator 0D(i: Int, d: Int): Int    ⟶    _ = 10*i + d
fun operator 0X(i: Int, d: Int): Int    ⟶    _ = 16*i + d
```

```
fun operator -(b: Byte):          Byte     ⟶    _ = -b
fun operator -(s: Short):         Short    ⟶    _ = -s
```

```
fun operator -(i: Int):             Int     ⟶   _ = -i
fun operator -(l: Long):            Long    ⟶   _ = -l
fun operator +(b: Byte,  c: Byte):  Byte    ⟶   _ = b + c
fun operator +(s: Short, t: Short): Short   ⟶   _ = s + t
fun operator +(i: Int,   j: Int):   Int     ⟶   _ = i + j
fun operator +(l: Long,  m: Long):  Long    ⟶   _ = l + m
fun operator -(b: Byte,  c: Byte):  Byte    ⟶   _ = b - c
fun operator -(s: Short, t: Short): Short   ⟶   _ = s - t
fun operator -(i: Int,   j: Int):   Int     ⟶   _ = i - j
fun operator -(l: Long,  m: Long):  Long    ⟶   _ = l - m
fun operator *(b: Byte,  c: Byte):  Byte    ⟶   _ = b * c
fun operator *(s: Short, t: Short): Short   ⟶   _ = s * t
fun operator *(i: Int,   j: Int):   Int     ⟶   _ = i * j
fun operator *(l: Long,  m: Long):  Long    ⟶   _ = l * m
fun operator /(b: Byte,  c: Byte):  Byte    ⟶   _ = b / c
fun operator /(s: Short, t: Short): Short   ⟶   _ = s / t
fun operator /(i: Int,   j: Int):   Int     ⟶   _ = i / j
fun operator /(l: Long,  m: Long):  Long    ⟶   _ = l / m
fun operator %(b: Byte,  c: Byte):  Byte    ⟶   _ = b % c
fun operator %(s: Short, t: Short): Short   ⟶   _ = s % t
fun operator %(i: Int,   j: Int):   Int     ⟶   _ = i % j
fun operator %(l: Long,  m: Long):  Long    ⟶   _ = l % m

fun operator ~(b: Byte):            Byte    ⟶   _ = ~b
fun operator ~(s: Short):           Short   ⟶   _ = ~s
fun operator ~(i: Int):             Int     ⟶   _ = ~i
fun operator ~(l: Long):            Long    ⟶   _ = ~l
fun operator &(b: Byte,  c: Byte):  Byte    ⟶   _ = b & c
fun operator &(s: Short, t: Short): Short   ⟶   _ = s & t
fun operator &(i: Int,   j: Int):   Int     ⟶   _ = i & j
fun operator &(l: Long,  m: Long):  Long    ⟶   _ = l & m
fun operator |(b: Byte,  c: Byte):  Byte    ⟶   _ = b | c
fun operator |(s: Short, t: Short): Short   ⟶   _ = s | t
fun operator |(i: Int,   j: Int):   Int     ⟶   _ = i | j
fun operator |(l: Long,  m: Long):  Long    ⟶   _ = l | m
fun operator ^(b: Byte,  c: Byte):  Byte    ⟶   _ = b ^ c
fun operator ^(s: Short, t: Short): Short   ⟶   _ = s ^ t
fun operator ^(i: Int,   j: Int):   Int     ⟶   _ = i ^ j
fun operator ^(l: Long,  m: Long):  Long    ⟶   _ = l ^ m

fun operator <<(b: Byte,  j: Int): Byte    ⟶   _ = b << j
fun operator <<(s: Short, j: Int): Short   ⟶   _ = s << j
fun operator <<(i: Int,   j: Int): Int     ⟶   _ = i << j
fun operator <<(l: Long,  j: Int): Long    ⟶   _ = l << j
fun operator >>(b: Byte,  j: Int): Byte    ⟶   _ = b >> j
fun operator >>(s: Short, j: Int): Short   ⟶   _ = s >> j
fun operator >>(i: Int,   j: Int): Int     ⟶   _ = i >> j
fun operator >>(l: Long,  j: Int): Long    ⟶   _ = l >> j

fun Byte (b: Byte):   Byte    ⟶   _ = b
fun Byte (i: Int):    Byte    ⟶   _ = (signed char)i
```

```
fun Short(s: Short):  Short    ⟶    _ = s
fun Short(i: Int):    Short    ⟶    _ = (short)i
fun Int  (b: Bool):   Int      ⟶    _ = (int)b
fun Int  (b: Char):   Int      ⟶    _ = (int)c
fun Int  (b: Byte):   Int      ⟶    _ = (int)b
fun Int  (s: Short):  Int      ⟶    _ = (int)s
fun Int  (i: Int):    Int      ⟶    _ = i
fun Int  (l: Long):   Int      ⟶    _ = (int)l
fun Int  (f: Float):  Int      ⟶    _ = (int)f
fun Int  (d: Double): Int      ⟶    _ = (int)d
fun Long (i: Int):    Long     ⟶    _ = (long)i
fun Long (l: Long):   Long     ⟶    _ = l

fun operator ==(b: Byte,  c: Byte):  Bool    ⟶    _ = b == c
fun operator ==(s: Short, t: Short): Bool    ⟶    _ = s == t
fun operator ==(i: Int,   j: Int):   Bool    ⟶    _ = i == j
fun operator ==(l: Long,  m: Long):  Bool    ⟶    _ = l == m
fun operator !=(b: Byte,  c: Byte):  Bool    ⟶    _ = b != c
fun operator !=(s: Short, t: Short): Bool    ⟶    _ = s != t
fun operator !=(i: Int,   j: Int):   Bool    ⟶    _ = i != j
fun operator !=(l: Long,  m: Long):  Bool    ⟶    _ = l != m
fun operator < (b: Byte,  c: Byte):  Bool    ⟶    _ = b <  c
fun operator < (s: Short, t: Short): Bool    ⟶    _ = s <  t
fun operator < (i: Int,   j: Int):   Bool    ⟶    _ = i <  j
fun operator < (l: Long,  m: Long):  Bool    ⟶    _ = l <  m
fun operator <=(b: Byte,  c: Byte):  Bool    ⟶    _ = b <= c
fun operator <=(s: Short, t: Short): Bool    ⟶    _ = s <= t
fun operator <=(i: Int,   j: Int):   Bool    ⟶    _ = i <= j
fun operator <=(l: Long,  m: Long):  Bool    ⟶    _ = l <= m
fun operator > (b: Byte,  c: Byte):  Bool    ⟶    _ = b >  c
fun operator > (s: Short, t: Short): Bool    ⟶    _ = s >  t
fun operator > (i: Int,   j: Int):   Bool    ⟶    _ = i >  j
fun operator > (l: Long,  m: Long):  Bool    ⟶    _ = l >  m
fun operator >=(b: Byte,  c: Byte):  Bool    ⟶    _ = b >= c
fun operator >=(s: Short, t: Short): Bool    ⟶    _ = s >= t
fun operator >=(i: Int,   j: Int):   Bool    ⟶    _ = i >= j
fun operator >=(l: Long,  m: Long):  Bool    ⟶    _ = l >= m

constr b: Byte ()            ⟶    (void)0
constr s: Short()            ⟶    (void)0
constr i: Int  ()            ⟶    (void)0
constr l: Long ()            ⟶    (void)0
constr b: Byte (c: Byte)     ⟶    b = c
constr s: Short(t: Short)    ⟶    s = t
constr i: Int  (j: Int)      ⟶    i = j
constr l: Long (m: Long)     ⟶    l = m

destr  b: Byte ()            ⟶    (void)0
destr  s: Short()            ⟶    (void)0
destr  i: Int  ()            ⟶    (void)0
destr  l: Long ()            ⟶    (void)0
```

```
axiom type Byte  case Int
axiom type Short case Int
axiom type Int   case Int
axiom type Long  case Int

proc operator =(out b: Byte,  c: Byte)     ⟶    b = c
proc operator =(out s: Short, t: Short)    ⟶    s = t
proc operator =(out i: Int,   j: Int)      ⟶    i = j
proc operator =(out l: Long,  m: Long)     ⟶    l = m
```

D.5 Floats

```
type Float     ⟶    float
type Double    ⟶    double

fun operator E-(s: String): String
    ⟶    ({char *_p = (char*)malloc(2 + strlen(s));
            _ = _p;
            *_p = '-';
            strcpy(_p + 1, s);
            })

fun operator Of(i: String, f: String, e: String): Double
    ⟶    ({int _l = strlen(i);
            int _m = 1 + _l + strlen(f);
            char *_a = (char*)malloc(2 + _m + strlen(e));
            strcpy(_a, i);
            _a[_l] = '.';
            strcpy(_a + 1 + _l, f);
            _a[_m] = 'e';
            strcpy(_a + 1 + _m, e);
            _ = atof(_a);
            free(_a);
            })

fun operator -(f: Float):             Float     ⟶    _ = -f
fun operator -(d: Double):            Double    ⟶    _ = -f
fun operator +(f: Float,  g: Float):  Float     ⟶    _ = f + g
fun operator +(d: Double, e: Double): Double    ⟶    _ = f + g
fun operator -(f: Float,  g: Float):  Float     ⟶    _ = f - g
fun operator -(d: Double, e: Double): Double    ⟶    _ = f - g
fun operator *(f: Float,  g: Float):  Float     ⟶    _ = f * g
fun operator *(d: Double, e: Double): Double    ⟶    _ = f * g
fun operator /(f: Float,  g: Float):  Float     ⟶    _ = f / g
fun operator /(d: Double, e: Double): Double    ⟶    _ = f / g

fun Float (i: Int):    Float     ⟶    _ = (float)i
fun Float (f: Float):  Float     ⟶    _ = f
fun Float (d: Double): Float     ⟶    _ = (float)d
```

```
fun Double(i: Int):    Double    ⟶    _ = (double)i
fun Double(f: Float):  Double    ⟶    _ = (double)f
fun Double(d: Double): Double    ⟶    _ = d

fun operator ==(f: Float,  g: Float):  Bool    ⟶    _ = f == g
fun operator ==(d: Double, e: Double): Bool    ⟶    _ = d == e
fun operator !=(f: Float,  g: Float):  Bool    ⟶    _ = f != g
fun operator !=(d: Double, e: Double): Bool    ⟶    _ = d != e
fun operator < (f: Float,  g: Float):  Bool    ⟶    _ = f <  g
fun operator < (d: Double, e: Double): Bool    ⟶    _ = d <  e
fun operator <=(f: Float,  g: Float):  Bool    ⟶    _ = f <= g
fun operator <=(d: Double, e: Double): Bool    ⟶    _ = d <= e
fun operator > (f: Float,  g: Float):  Bool    ⟶    _ = f >  g
fun operator > (d: Double, e: Double): Bool    ⟶    _ = d >  e
fun operator >=(f: Float,  g: Float):  Bool    ⟶    _ = f >= g
fun operator >=(d: Double, e: Double): Bool    ⟶    _ = d >= e

constr f: Float ()            ⟶    (void)0
constr d: Double()            ⟶    (void)0
constr f: Float (g: Float)    ⟶    f = g
constr d: Double(e: Double)   ⟶    d = e

destr  f: Float ()            ⟶    (void)0
destr  d: Double()            ⟶    (void)0

proc operator =(out f: Float,  g: Float)     ⟶    f = g
proc operator =(out d: Double, e: Double)    ⟶    d = e
```

D.6 Addresses

```
type Address    ⟶    void*

val nil: Address    ⟶    (void*)0

fun operator +(a: Address, i: Int): Address
    ⟶    _ = (void*)((char*)a + i)
fun operator -(a: Address, i: Int): Address
    ⟶    _ = (void*)((char*)a - i)
fun operator -(a: Address, b: Address): Int
    ⟶    _ = (char*)a - (char*)b

fun Address(a: Address): Address    ⟶    _ = a

fun operator ==(a: Address, b: Address): Bool    ⟶    _ = a == b
fun operator !=(a: Address, b: Address): Bool    ⟶    _ = a != b
fun operator < (a: Address, b: Address): Bool    ⟶    _ = a <  b
fun operator <=(a: Address, b: Address): Bool    ⟶    _ = a <= b
fun operator > (a: Address, b: Address): Bool    ⟶    _ = a >  b
fun operator >=(a: Address, b: Address): Bool    ⟶    _ = a >= b

constr a: Address()             ⟶    (void)0
constr a: Address(b: Address)   ⟶    a = b
```

```
destr  a: Address()                 ⟶    (void)0

axiom type Address address Address

proc alloc (out   a: Address, i: Int)    ⟶    a = malloc(i)
proc delete(inout a: Address)            ⟶    free(a)

proc operator =(out a: Address, b: Address)    ⟶    a = b
```

D.7 Records and Unions

For any record or union type T and any of its components (alternatives) c of type C, the following operations are built-in:

```
constr t: T()    ⟶    (void)0

destr  t: T()    ⟶    (void)0

reffun operator .c(inref t: T): C    ⟶    t.c
```

Appendix E

Example

This appendix contains a simple example of a Mona module and its usage. The module implements lists of built-in integers. In detail, the following is contained:

- The *Mona sources* required for building the module, i.e., the *specification* of the module, the (parameter-less) *functor* implementing the specification, as well as the *module* unit describing the module as instance of the functor.
- The *code generated* when compiling the module unit, i.e., the C++ *header file* for the module, together with the files containing the *structure definitions* and *inline definitions*, and the C++ *source file* from which the object code for the module is generated by g++.
- The *export interface* produced when exporting the module, i.e., the *export header file* and the *make include file* giving the locations of the needed object files. The identifiers defined in the export header file do not have the module name prepended, since 'unqualified' export was requested.
- The *usage* of the module is demonstrated by a C++ *main source file* defining a `main` function that uses the items exported by the module, a *make file* for generating a linked executable from this source file and the module object code, and the *output* from running this executable.

E.1 Mona Sources

E.1.1 Specification

```
spec List let {
  type Bool  builtin Bool
  type Int   builtin Int

  fun operator !(b: Bool): Bool
} in {
  type List

  fun isEmpty(l: List): Bool
```

```
  reffun operator .first(inref l: List): Int
    input !isEmpty(l)
  reffun operator .rest (inref l: List): List
    input !isEmpty(l)

  constr c: List(l: List)
  destr  l: List()

  val empty                : List
  fun cons(i: Int, l: List): List

  proc operator =(out o: List, l: List)

  proc print(l: List)
}
```

E.1.2 Functor

```
functor List: List {
  //
  // truth values
  //
  type Bool  builtin Bool

  val false: Bool  builtin false
  val true : Bool  builtin true

  fun Bool      (b: Bool): Bool  builtin Bool
  fun operator !(b: Bool): Bool  builtin operator ~

  proc operator =(out o: Bool, b: Bool)  builtin operator =

  //
  // integers
  //
  type Int  builtin Int

  val #0                 : Int  builtin #0
  fun operator 0D(i: Int): Int  builtin Int
  fun operator 0d(i: Int): Int  builtin Int

  proc operator =(out o: Int, i: Int)  builtin operator =

  proc print(i: Int)  extern printInt

  //
  // characters
  //
  type Char  builtin Char

  val '(': Char  builtin '('
  val ',': Char  builtin ','
  val ' ': Char  builtin ' '
  val ')': Char  builtin ')'

  proc print(c: Char)  extern printChar

  //
  // addresses
  //
  type Address  builtin Address
```

```
val nil: Address  builtin nil

fun operator ==(a: Address, b: Address): Bool  builtin operator ==

proc alloc (out   a: Address, i: Int)  builtin alloc
proc delete(inout a: Address)          builtin delete

proc operator =(out o: Address, a: Address)  builtin operator =

//
// pairs
//
type List  forward
type PairRepr = Record[
  first: Int
  rest : List
]
type Pair new PairRepr

reffun operator .first(inref r: PairRepr): Int   builtin operator .first
reffun operator .first(inref p: Pair): Int   alias operator .first(repr)
reffun operator .rest (inref r: PairRepr): List  builtin operator .rest
reffun operator .rest (inref p: Pair): List  alias operator .rest (repr)

constr c: Pair()  inline
  constr repr c: PairRepr();
constr c: List(l: List)  forward
constr c: Pair(p: Pair) {
  constr c: Pair();
  constr c.first: Int (p.first);
  constr c.rest : List(p.rest);
}

destr l: List()  forward
destr p: Pair() {
  destr p.rest : List();
  destr p.first: Int ();
  destr repr p: PairRepr();
}

fun pair(i: Int, l: List) p: Pair {
  constr p: Pair();
  constr p.first: Int (i);
  constr p.rest : List(l);
}

proc operator =(out o: List, l: List)  forward
proc operator =(out o: Pair, p: Pair) {
  o.first = p.first;
  o.rest  = p.rest;
}

proc print(l: List)  forward
proc print(p: Pair) {
  print(p.first); print(','); print(' '); print(p.rest);
}

//
// lists
//
type List  new Address

fun isEmpty(l: List): Bool  inline
  return (repr l) == nil;
```

```
  reffun operator .first(inref l: List): Int  inline
    return (Pair @ repr l).first;

  reffun operator .rest (inref l: List): List  inline
    return (Pair @ repr l).rest;

  constr c: List(l: List)  inline {
    if (isEmpty(l)) constr repr c: Address(nil);
    else {
      constr  repr c: Address();
      alloc(repr c, sizeof Pair);
      constr Pair @ repr c: Pair(Pair @ repr l);
    }
  }

  destr l: List()  inline {
    if (!isEmpty(l)) destr Pair @ repr l: Pair();
    delete(repr l);
    destr repr l: Address();
  }

  val empty: List
    init  constr repr empty: Address(nil);
    term  destr  repr empty: Address();

  fun cons(i: Int, l: List) c: List  inline {
    constr repr c: Address();
    alloc(repr c, sizeof Pair);
    constr Pair @ repr c: Pair(pair(i, l));
  }

  proc operator =(out o: List, l: List)  inline {
    if (isEmpty(o) && !isEmpty(l)) {
      alloc(repr o, sizeof Pair);
      constr Pair @ repr o: Pair(Pair @ repr l);
    } else if (!isEmpty(l)) {
      Pair @ repr o = Pair @ repr l;
    } else {
      destr Pair @ repr o: Pair();
      delete(repr o); repr o = nil;
    }
  }

  proc print(l: List)  inline {
    print('('); if (!isEmpty(l)) print(Pair @ repr l); print(')');
  }
}
```

E.1.3 Module

```
module List: List = List[]
```

E.2 Generated Code

E.2.1 Header File

```
#ifndef List_O_h
#define List_O_h
```

```
/*************************/
/* structure definitions */
/*************************/
#include "/home/hpgp/test/functor/List/instances/0/List_0.t"

#ifdef __cplusplus
  extern "C" {
#endif

/*********************/
/* implemented items */
/*********************/
#ifdef List_0_cc
  #define val
#else
  #define val const
#endif

typedef int _0_List_0;   /* Bool */
static int const _1_List_0 = 0;  /* false */
static int const _2_List_0 = 1;  /* true */
#define _3_List_0                        /* Bool */
#define _4_List_0(o, x) ((o) = !(x))     /* __op_excl */
#define _5_List_0(x, y) ((x) = (y))      /* __op_assign */
typedef int _6_List_0;   /* Int */
static int const _7_List_0 = 0;  /* __dig_0 */
#define _8_List_0                        /* __op_0D */
#define _9_List_0                        /* __op_0d */
#define _10_List_0(x, y) ((x) = (y))     /* __op_assign */
void printInt(int);
#define _11_List_0(_0) printInt((_0))    /* print */
typedef char _12_List_0;         /* Char */
static char const _13_List_0 = '(';      /* __chr_lparen */
static char const _14_List_0 = ',';      /* __chr_comma */
static char const _15_List_0 = ' ';      /* __chr_spc */
static char const _16_List_0 = ')';      /* __chr_rparen */
void printChar(char);
#define _17_List_0(_0) printChar((_0))   /* print */
typedef void* _18_List_0;        /* Address */
static void* const _19_List_0 = 0;       /* nil */
#define _20_List_0(o, x, y) ((o) = (x) == (y))   /* __op_eq */
#define _21_List_0(x, i) ((x) = malloc(i))       /* alloc */
#define _22_List_0       free    /* delete */
#define _23_List_0(x, y) ((x) = (y))     /* __op_assign */
typedef _Record_0_List_0 _24_List_0;     /* PairRepr */
typedef _Record_0_List_0 _25_List_0;     /* Pair */
#define _26_List_0(x) (x).first  /* __op_dot_first */
#define _27_List_0(_0) _26_List_0((_0))  /* __op_dot_first */
#define _28_List_0(x) (x).rest   /* __op_dot_rest */
#define _29_List_0(_0) _28_List_0((_0))  /* __op_dot_rest */
inline void __30_List_0(_Record_0_List_0& _c);
#define _30_List_0(_c) __30_List_0((_Record_0_List_0&)(_c))                  \
    /* (constructor) */
void __31_List_0(_Record_0_List_0& _c, _Record_0_List_0 const& _p);
#define _31_List_0(_c, _p) __31_List_0((_Record_0_List_0&)(_c),              \
    (_Record_0_List_0 const&)(_p))       /* (constructor) */
void __32_List_0(_Record_0_List_0& _p);
#define _32_List_0(_p) __32_List_0((_Record_0_List_0&)(_p))                  \
    /*(destructor) */
void __33_List_0(_Record_0_List_0&, int _i, void* _l);
#define _33_List_0(_, _i, _l) __33_List_0((_Record_0_List_0&)(_), (_i), (_l))   \
    /* pair */
```

```
void __34_List_0(_Record_0_List_0& _o, _Record_0_List_0 const& _p);
#define _34_List_0(_o, _p) __34_List_0((_Record_0_List_0&)(_o),                \
    (_Record_0_List_0 const&)(_p))       /* __op_assign */
void __35_List_0(_Record_0_List_0 const& _p);
#define _35_List_0(_p) __35_List_0((_Record_0_List_0 const&)(_p))              \
    /* print */
typedef void* _36_List_0;        /* List */
inline int __37_List_0(void* _l);
#define _37_List_0(_, _l) ((_) = __37_List_0((_l)))      /* isEmpty */
inline int* __38_List_0(void* _l);
#define _38_List_0(_l) (*__38_List_0((_l)))      /* __op_dot_first */
inline void** __39_List_0(void* const& _l);
#define _39_List_0(_l) (*__39_List_0((_l)))      /* __op_dot_rest */
inline void* __40_List_0(void* _l);
#define _40_List_0(_, _l) ((_) = __40_List_0((_l)))      /* (constructor) */
inline void __41_List_0(void*& _l);
#define _41_List_0(_l) __41_List_0((_l))        /* (destructor) */
extern void* val _42_List_0;     /* empty */
inline void* __43_List_0(int _i, void* _l);
#define _43_List_0(_, _i, _l) ((_) = __43_List_0((_i), (_l)))    /* cons */
inline void __44_List_0(void*& _o, void* _l);
#define _44_List_0(_o, _l) __44_List_0((_o), (_l))       /* __op_assign */
inline void __45_List_0(void* _l);
#define _45_List_0(_l) __45_List_0((_l))        /* print */

#undef val

/******************************/
/* initialization/termination */
/******************************/
void _init_List_0();
void _term_List_0();

/******************/
/* runtime systen */
/******************/
void*    malloc(int);
void     free  (void*);
unsigned strlen(char const*);
char*    strcpy(char*, char const*);

#ifdef __cplusplus
  }
#endif

/***********************/
/* parameter instances */
/***********************/

/**********************/
/* inline definitions */
/**********************/
#include "/home/hpgp/test/functor/List/instances/0/List_0.i"

#endif
```

E.2.2 Structure Definitions

```
struct _Record_0_List_0 {
  int first;
  void* rest;
};
```

E.2.3 Inline Definitions

```
#line 1 "/home/hpgp/test/functor/List/source"
#pragma interface "/home/hpgp/test/functor/List/source"
#line 56
extern inline void __30_List_0(_Record_0_List_0& _c)
{ ; }
#line 97
extern inline int __37_List_0(void* _l) {int _;
{ {_ = (_l == _19_List_0);return _;} } }
extern inline int* __38_List_0(void* _l) {int* _;
{ {{int* _0;{_Record_0_List_0* _1;_1 = (_Record_0_List_0*)_l;_0 =            \
    &(*_1).first;}_ = &(*_0);}return _;} } }
extern inline void** __39_List_0(void* const& _l) {void** _;
{ {{void** _0;{_Record_0_List_0* _1;_1 = (_Record_0_List_0*)_l;_0 =          \
    &(*_1).rest;}_ = &(*_0);}return _;} } }
extern inline void* __40_List_0(void* _l) {void* _c;
{
  if (({int __;{int _0;_37_List_0(_0, _l);__ = _0;;}__;}))
    _c = _19_List_0;
  else
    {
      ;
      {int _0;_0 = sizeof(_Record_0_List_0);_c = malloc(_0);}
      {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_c;{_Record_0_List_0* _1;_1 \
          = (_Record_0_List_0*)_l;_31_List_0((*_0), (*_1));}}
    }
} return _c;}
extern inline void __41_List_0(void*& _l)
{
  if (({int __;{int _0;{int _1;_37_List_0(_1, _l);_0 = (!_1);;}__ = _0;;}__;}))
    {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_l;_32_List_0((*_0));}
  free(_l);
  ;
}
#line 126
extern inline void* __43_List_0(int _i, void* _l) {void* _c;
{
  ;
  {int _0;_0 = sizeof(_Record_0_List_0);_c = malloc(_0);}
  {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_c;_33_List_0((*_0), _i, _l);}
} return _c;}
extern inline void __44_List_0(void*& _o, void* _l)
{
  if (({int __;{int __0;{int _0;{int _1;_37_List_0(_1, _o);__0 = _1;;}if (!__0) \
      goto _0;{int _1;{int _2;_37_List_0(_2, _l);_1 = (!_2);;}_0 =              \
      _1;goto _1;_0:_0 = 0;_1:;__ = _0;}if (__0) goto _2;goto                   \
      _3;_2:;}_3:;}__;}))
    {
      {int _0;_0 = sizeof(_Record_0_List_0);_o = malloc(_0);}
      {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_o;{_Record_0_List_0* _1;_1 \
          = (_Record_0_List_0*)_l;_31_List_0((*_0), (*_1));}}
    }
  else if (({int __;{int _0;{int _1;_37_List_0(_1, _l);_0 = (!_1);;}__ =        \
      _0;;}__;}))
    {
      {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_o;{_Record_0_List_0* _1;_1 \
          = (_Record_0_List_0*)_l;_34_List_0((*_0), (*_1));}}
    }
  else
    {
      {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_o;_32_List_0((*_0));}
```

```
      free(_o);
      _o = _19_List_0;
    }
}
extern inline void __45_List_0(void* _l)
{
  _17_List_0(_13_List_0);
  if (({int __;{int _0;{int _1;_37_List_0(_1, _l);_0 = (!_1);;}__ = _0;;}__;}))
    {_Record_0_List_0* _0;_0 = (_Record_0_List_0*)_l;_35_List_0((*_0));}
  _17_List_0(_16_List_0);
}
```

E.2.4 Source File

```
#define List_0_cc
#pragma implementation "/home/hpgp/test/functor/List/source"
#include "/home/hpgp/test/functor/List/instances/0/List_0.h"

#line 1 "/home/hpgp/test/functor/List/source"
#line 59
void __31_List_0(_Record_0_List_0& _c, _Record_0_List_0 const& _p)
{
  _30_List_0(_c);
  {int* _0;_0 = &_c.first;{int const* _1;_1 = &_p.first;(*_0) = (*_1);}}
  {void** _0;_0 = &_c.rest;{void* const* _1;_1 = &_p.rest;_40_List_0((*_0),     \
      (*_1));}}
}
#line 66
void __32_List_0(_Record_0_List_0& _p)
{
  {void** _0;_0 = &_p.rest;_41_List_0((*_0));}
  ;
  ;
}
void __33_List_0(_Record_0_List_0& _p, int _i, void* _l)
{
  _30_List_0(_p);
  {int* _0;_0 = &_p.first;(*_0) = _i;}
  {void** _0;_0 = &_p.rest;_40_List_0((*_0), _l);}
}
#line 79
void __34_List_0(_Record_0_List_0& _o, _Record_0_List_0 const& _p)
{
  {int* _0;{int const* _1;_0 = &_o.first;_1 = &_p.first;(*_0) = (*_1);}}
  {void** _0;_0 = &_o.rest;{void* const* _1;_1 = &_p.rest;_44_List_0((*_0),     \
      (*_1));}}
}
#line 85
void __35_List_0(_Record_0_List_0 const& _p)
{
  {int const* _0;_0 = &_p.first;_11_List_0((*_0));}
  _17_List_0(_14_List_0);
  _17_List_0(_15_List_0);
  {void* const* _0;_0 = &_p.rest;_45_List_0((*_0));}
}
#line 121
void* _42_List_0;
static int initialized = 0;
#line 1
void _init_List_0() {if (!initialized++)
```

```
#line 123
{ _42_List_0 = _19_List_0; } }
#line 1
void _term_List_0() {if (!--initialized)
#line 125
{ ; } }
```

E.3 Export Interface

E.3.1 Export Header File

```
#ifndef List_h
#define List_h

/**********************/
/* module header file */
/**********************/
#include "/home/hpgp/test/functor/List/instances/0/List_0.h"

/************************************/
/* module initialization/termination */
/************************************/
#define init()   (_init_List_0())
#define term()   (_term_List_0())

/*****************/
/* exported items */
/*****************/

/* types and their constructors/destructors: */
/* ---------------------------------------- */
#define _type_List       _36_List_0      /* type List */
#define _constr_List     _40_List_0      /* constr: List(List) */
#define _destr_List      _41_List_0      /* destr: List() */

/* procedures, functions, values and variables: */
/* ------------------------------------------- */
#define cons     _43_List_0      /* fun(builtin Int, List): List */
#define empty    _42_List_0      /* val: List */
#define __op_dot_rest    _39_List_0      /* reffun(inref: List): List */
#define print    _45_List_0      /* proc(List) */
#define __op_assign      _44_List_0      /* proc(out: List, List) */
#define isEmpty  _37_List_0      /* fun(List): builtin Bool */
#define __op_dot_first   _38_List_0      /* reffun(inref: List): builtin Int */

/* contents of exported modules: */
/* ---------------------------- */

#endif
```

E.3.2 Make Include File

```
LIST_OBJS=\
  /home/hpgp/test/functor/List/instances/0/List_0.o
```

E.4 Usage

E.4.1 Main Source File

```
#include <stdio.h>
#include "List.h"

void printInt (int  i) { printf("%i", i); }
void printChar(char c) { printf("%c", c); }

int main() {
  init();

  _type_List single;
  _type_List duple;
  _type_List triple;

  print(empty);  printf("\n\n");

  cons(single, 1, empty);

  print(empty);  printf("\n");
  print(single); printf("\n\n");

  cons(duple,  2, single);

  print(empty);  printf("\n");
  print(single); printf("\n");
  print(duple);  printf("\n\n");

  cons(triple, 3, duple);

  print(empty);  printf("\n");
  print(single); printf("\n");
  print(duple);  printf("\n");
  print(triple); printf("\n\n");

  _type_List copy;

  _constr_List(copy, triple);
  print(copy); printf(" -> ");
  __op_assign(copy, single);
  print(copy); printf(" -> ");
  __op_dot_first(copy) = 4; __op_assign(__op_dot_rest(copy), triple);
  print(copy); printf("\n\n");

  _destr_List(copy);

  print(empty);  printf("\n");
  print(single); printf("\n");
  print(duple);  printf("\n");
  print(triple); printf("\n");

  _destr_List(triple);
  _destr_List(duple);
  _destr_List(single);

  term();

  return 0;
}
```

E.4.2 Make File

```
include List.mk

list: List.h list.cc $(LIST_OBJS)
	gcc -g -o list list.cc $(LIST_OBJS)
```

E.4.3 Output

```
()

()
(1, ())

()
(1, ())
(2, (1, ()))

()
(1, ())
(2, (1, ()))
(3, (2, (1, ())))

(3, (2, (1, ()))) -> (1, ()) -> (4, (3, (2, (1, ()))))

()
(1, ())
(2, (1, ()))
(3, (2, (1, ())))
```

References

[ANS83] United States Government Printing Office. *Reference Manual for the Ada Programming Language*, 1983. Military Standard ANSI/MIL-STD-1815A-1983.

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

[Ber93] Dave Berry. Lessons from the design of a Standard ML library. *Journal of Functional Programming*, 3(4):527–552, October 1993.

[Bis95] Sandip K. Biswas. Higher-order functors with transparent signatures. In *POPL '95*, pages 154–163, San Francisco, California, January 22–25, 1995. ACM Press, New York.

[CH+94] Olga Caprotti, Hong Hong, et al. On C++ polymorphism for the STURM library. STURM Project Internal Note 94-001, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, 1994.

[CM94] Pierre Cregut and David B. MacQueen. An implementation of higher order functors. In *ACM SIGPLAN Workshop on ML and its Applications*, Orlando, Florida, June 25–26, 1994. ACM, New York.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[HN+94] Hoon Hong, Andreas Neubacher, et al. The STURM Library Manual: A C++ library for symbolic computation. Technical Report 94-30, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, April 1994.

[HS96] Hoon Hong and Wolfgang Schreiner. HPGP: High-performance generic programming for computational mathematics by compile-time instantiation of higher-order functors. Technical Report 96-10, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, April 1996.

[ISO94] International Organization for Standardization. *Ada Reference Manual: Language and Standard Libraries (Version 6.0)*, December 1994. International Standard ISO/IEC 8652:1995(E).

[Jen92] Richard D. Jenks. *Axiom: The Scientific Computation System.* Springer, Berlin, 1992.

[KR88] Brian Kernighan and Dennis Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, NJ, 2nd edition, 1988.

[Lan96] Werner Landerl. Ida—the implementation language. Project Report HPGP-96-6, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, 1996. http://www.risc.uni-linz.ac.at/projects/basic/hpgp/reports/96-6.

[Ler95] Xavier Leroy. Applicative functors and fully transparent higher-order modules. In *POPL '95*, pages 142–153, San Francisco, CA, January 22–25, 1995. ACM Press.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc.* O'Reilly, Sebastopol, CA, 2nd edition, 1992.

[MS96] David R. Musser and Atul Saini. *STL Tutorial & Reference Guide.* Addison-Wesley, Reading, MA, 1996.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Nel91] Greg Nelson. *Systems Programming with Modula-3.* Prentice-Hall, Englewood Cliffs, NJ, 1991.

[Scr88] Computer Algebra Group, Mathematical Sciences Department, IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY. *The Scratchpad II Computer Algebra System: Interactive Environment Users Guide, Draft 1.2*, September 1988.

[SDLMS98] Wolfgang Schreiner, Werner Danielczyk-Landerl, Mircea Marin, and Wolfgang Stöcher. HPGP User and Reference Manual. Technical report, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, 1998. To appear.

[Stö97] Wolfgang Stöcher. Designing and prototyping a functor language using denotational semantics. Master's thesis, Johannes Kepler Universität, Linz, Austria, 1997. Also: Technical Report 97-37, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, 1997.

[Str88] Bjarne Stroustrup. Parameterized types for C++. In *USENIX C++ Conference*, Denver, Colorado, October 1988. Also: *USENIX Computer Systems*, 2(3), Winter 1989.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, MA, 1994.

[Str97] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, MA, 3rd edition, 1997.

[WB93] Wolfgang Windsteiger and Bruno Buchberger. GRÖBNER: A library for computing Gröbner bases based on SACLIB. Technical Report 93-72, Research Institute for Symbolic Computation (RISC-Linz), Johannes Kepler Universität, Linz, Austria, 1993.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, November 1977.

[Wir85] Niklaus Wirth. *Programming in Modula-2*. Texts and Monographs in Computer Science. Springer, Berlin, 3rd edition, 1985.