UNIVERSITÉ CATHOLIQUE DE LOUVAIN
ÉCOLE POLYTECHNIQUE DE LOUVAIN
DÉPARTEMENT D'INGÉNIERIE INFORMATIQUE

A modular Oz compiler for the new 64-bit Mozart virtual machine

Supervisor: Peter Van Roy
Readers: Sébastien Doeraene
Pierre Schaus

Mémoire présenté en vue de l'obtention du grade de master ingénieur civil en informatique option software engineering and programming systems par Raphaël Bauduin

Louvain-La-Neuve
Année académique 2012-2013

I would like to thank

- Peter Van Roy who trusted me enough to let me work on this compiler
- Pierre Schaus who kindly accepted to be a reader of this report
- Sébastien Doeraene who patiently supported my efforts, answering even the dumbest questions

Contents

Chapter 1

Introduction

This report documents the development of a compiler for the Oz language. Oz was first designed in 1991. It is a multi-paradigm language, supporting imperative, object-oriented, functional, logic and constraint programming [OzOv].

The Mozart Programming System is a multi-platform implementation of the Oz programming language[Moz], the last stable release dating from 2008.

It was deemed necessary to refresh the platform, and a new implementation was started, which we will refer to by Mozart2.

1.1 Initial State

A new virtual machine had been developed for Mozart2. The compiler targeting the Mozart1 virtual machine had been adapted to target the new virtual machine, and a boot compiler developed in Scala was used to compile it for the new virtual machine.

This was seen as a temporary solution though, and a new compiler needed to be developed.

1.2 Goal and Scope

The new compiler was to be developed in Oz. There were four goals set for the compiler:

1. compile the whole language so that it can replace the current compiler in Mozart 2

2. generate quality code, exploiting the capabilities of the virtual machine

3. easy to understand and modular code

4. the compiler should also be extensible so that, for example, support for a new instruction can be added without the recompilation of the compiler.

Although the code is extensively commented, this report contributes to the third goal. The first three goals have been reached or approached, only the fourth proved to be to ambitious for this work.

1.3 Contributions of this work

This work has the goal to lay a solid foundation for future developments. The whole code was written from scratch. Nearly the whole Oz language is supported by the compiler, and the code is clearly structured and fully documented. A flexible and easy to use test infrastructure has been set up. This test infrastructure has been used to write 439 tests scripts, performing more than 1000 output checks. An Oz script runner is also included, compiling the Oz source file and outputting AST at each transformation step for easy debug. Compiling a file to a `.ozf` is also possible.

1.4 Source Code

The whole source code for the compiler, the tests as well as this report is available at `http://www.github.com/raphinou/oz-compiler`. It can be downloaded as a zip file or checked out using git with the command
`git clone git://github.com/raphinou/oz-compiler.git`.

Chapter 2

Infrastructure

2.1 Virtual Machine

The Mozart2 virtual machine is a bytecode vm developed from scratch, with extensive support for the Oz language's concepts. The code to be generated by the compiler should use these capabilities and possibly be different in some cases from the code generated by the Mozart1 compiler.

2.1.1 Registers

The Mozart2 virtual machine works with four kind of registers. The description of the registers uses the concept of abstractions, which is precisely defined in the next section. It is however sufficient for the understanding of this section to equate the concept of abstraction with the concept of procedural value.

X registers X registers are work registers that should not be used to permanently store values as their content is undefined after a call. They are thread-specific and caller-saved, meaning it is the caller's responsibility to save the value of an X register if it will be reused. X registers are for example used to pass the arguments to calls. However, the values in these registers are undefined after the call so they should be considered as lost. X registers are currently indexed by an unsigned 16 bit integer, but this might change in the future to be indexed by a signed integer.

Y registers are specific to the activation frame, and are used as locally persisting registers. They can be allocated only once per frame.

G registers are holding global variables, i.e. variables that are accessed by the abstraction, but that are not locally declared. They are specific to the abstraction.

K registers are holding constant values, be it integers, floats or records. They are specific to the abstraction's code area, which we will analyse in the next section. A constant value is a value known by the compiler at compile time, which can be an unbound variable

2.1.2 Abstractions

An Oz procedure is represented in the virtual machine by what is called an abstraction. An Oz program itself is implicitly placed in what is called a top level abstraction. An abstraction holds references to G registers and to its code area. This code area has references to its K registers and the code itself, as illustrated in Figure 2.1.

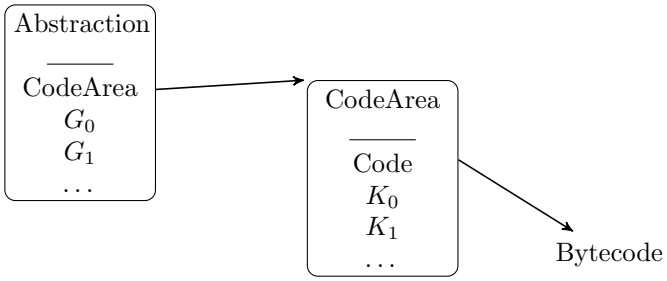

Figure 2.1: Internals of the virtual machine

Capturing the global variables at the abstraction level is what provides closures: a procedure captures the environment it was defined in. The CodeArea is not stored in the abstraction itself, but is referenced from it, simply because the same CodeArea can be referenced from multiple abstractions. The example from [CTMCPFigs] in Figure 2.2 illustrates this.

```
fun {Sqrt X}
   fun {Improve Guess}
     (Guess + X/Guess) / 2.0
   end
   fun {GoodEnough Guess}
     {Abs X-Guess*Guess}/X < 0.00001
   end
   fun {SqrtIter Guess}
      if {GoodEnough Guess} then Guess
      else
         {SqrtIter {Improve Guess}}
      end
   end
   Guess=1.0
in
   {SqrtIter Guess}
end
```

Figure 2.2: Functions sharing the same CodeArea

The method Sqrt computes the square root of a number by Newton's method. Each time Sqrt is called, it defines three local functions. The two first reference the variable X, which is a global variable for them, and which has probably a different value at each call of Sqrt. However, the code area for each function is the same for every call of Sqrt, and the virtual machine can

simply create a new abstraction referencing the already existing code area and defining the global it needs. Often, code areas are created at compile time, and the abstractions at run time.

If we have 2 simultaneous calls to Sqrt, for example Sqrt 2 and Sqrt 5, we would end up with the structure of Figure 2.3 in the virtual machine.

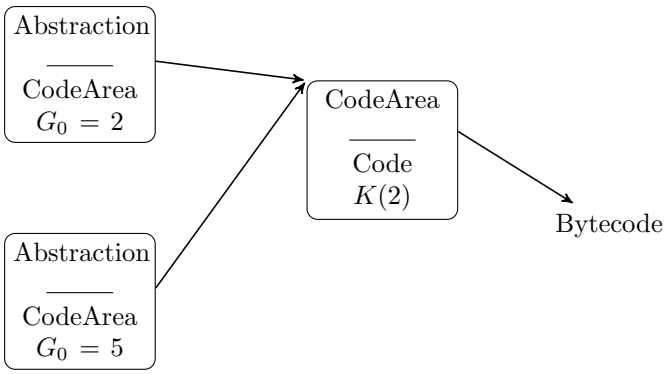

Figure 2.3: Multiple Abstractions referring to the same CodeArea

2.2 Target Language

The compiler generates code that can be passed to the assembler targeting the virtual machine. This assembler language has solid support for the virtual machine concepts as we will see. Appendix B gives a reference list of the opcodes available.

The core of the virtual machine is the emulator loop which interprets the bytecode. This loop is found in the file `emulate.cc` of the virtual machine [Moz2vmsrc] [1].

The assembler accepts an Oz list of instructions that are Oz tuples. This list of opcodes is actually further transformed by the assembler before it gets its definitive form and is fed to the virtual machine.

One transformation applied by the assembler is to translate generic operations on register to operations specific to the registers manipulated.

For example, we will present the `move(src dst)` opcode taking a source register and a destination register. This opcode is actually specialised according to the source and destination register types. Source register can be X, Y, G or K registers. Destination registers can be X or Y registers. As a consequence, `move` is actually translated into 8 specialised operations. The resolution is done by the assembler with a function `ResolveOverload` [Moz2src][2] which will actually resolve all overloaded opcodes that will be covered in this section.

2.2.1 Register Operations

Registers access instructions are proper to each kind of register. To access the nth G X Y registers, the instruction will respectively be `g(n)`, `x(n)`, `y(n)`.

[1] http://git.io/5CcN2A
[2] http://git.io/XLk5wQ

K registers are used to store constants. The assembler abstracts away the actual indexing of K registers. It expects values in `k(V)` tuples, and assigns indices to K registers itself. The constant 2 will thus appear in the code as `k(2)`, the constant record `r(a b c)` is denoted in the code as `k(r(a b c))`.

As mentioned earlier, Y registers are persisting for the lifespan of the current frame. Their number has to be determined when the frame is created, and all needed Y registers are allocated at once with the instruction `allocateY(n)` where n is the number of Y registers used in this frame. Y registers are deallocated when their owning frame is disposed of, i.e. upon execution of one of `return`, `tailCall` or `tailSendMsg`.

X registers are work registers, and if a value in an X register needs to be accessed several time across calls, it has to be saved in a Y register. This is done with a move instruction: `move(source_reg destination_reg)`. As mentioned earlier, source can be X, Y, G or K registers. Destination can be X or Y registers.

Unification is supported with the instruction `unify(LHS RHS)` where `LHS` and `RHS` are X, Y, G, or K registers. Internally, and because unification is commutative, this does not result in 16 but 10 specialised opcodes. For example, `moveXY` is defined, but not its mirror operator `moveYX`. It is the assembler's `ResolveOverload` function that will pick the available specialised operator and possibly switch the arguments order.

Two consecutive `move` operations on registers X and/or Y registers are transformed by the assembler in a `moveMove` operation, transforming

```
move(s1 d1) move(s2 d2)
```

into

```
moveMove(s1 d1 s2 d2)
```

Four different situations can occur as both source and destination can be X or Y registers resulting in 4 specialised operations `moveMoveXYXY`, `moveMoveYXYX`, `moveMoveYXXY`, `moveMoveXYYX`.

2.2.2 Variables

The opcode `createVar(R)` assigns to register `R`, which is an X or Y register, a new unbound variable. The assembler also translates it to `createVarX` and `createVarY`.

`createVarMove(r(i) x(j))` is an opcode that will assign an unbound variable to register `r(i)` (X or Y), immediately followed by a `move` to register `x(j)` (always an X register). This is very handy when an unbound variable is passed as argument to a procedure, to be reused after the call, as in

```
{MyProc ?R}
{Show R}
```

The variable `R` has to be in a Y register to be reused after the call, but it has to be passed as argument to the call in an X register. `createVarMove` does both operations in one opcode. Again, `createVarMove` is actually specialised in `createVarMoveX` or `createVarMoveY` if the first register is respectively an X or a Y register.

2.2.3 Jumps

Code positions can be identified by labels, which need to be atoms or names. A label `L` is encoded in the code as a tuple `lbl(L)`. Labels are required for jumps in the code. Jumps will move the execution to the label they have as argument. For example `branch(endLabel)` will jump to the position in the code identified by the label `endLabel`.

Conditional jumps will look at the value in an X register, and if this value is not true, it will jump to the code at the label it was given. That instruction also takes a label locating error handling code. Here is an example of conditional jump that will look at the value in X register 0:

```
condBranch(x(0) elseLabel errorLabel)
```

If the value in x(0) is true, it continues execution. If it is false it jumps to the position in the code identified by the label `elseLabel`. If an error occurs in the test, it jumps to the label `errorLabel`. This is illustrated in Figure 2.4. If the value in register `x(0)` is not tested successfully as a boolean, execution will jump to `lbl(1)` and raise an exception. If the condition (the value in `x(0)`) is true, the code will execute `{Show 'true'}`, else it will execute `{Show false}`.

```
                condBranch(x(0) 2 1)
                move(k('true') x(0))
                call(k(<P/1 Show>) 1)
                branch(3)
lbl(1)          move(k(badBooleanInIf) x(0))
                tailCall(k(<P/1 Exception.raiseError>) 1)
lbl(2)          move(k('false') x(0))
                call(k(<P/1 Show>) 1)
lbl(3)          return
```

Figure 2.4: Conditional branch example

2.2.4 Calls

Calls can be made with `call(R Args)`, where R is an X, Y, G or K register. identifying what to call (procedure or builtin) and Args is the number of arguments passed to the callee. As per the calling convention of the vm bytecode, the arguments of the call have to be placed in the X registers from 0 to Args-1 with move instructions before the call. The complete code for a call to a callee located in register Y 3 with three arguments which values come from Y registers 4 to 6 will thus be of the form illustrated in Figure 2.5.

A call done as last instruction of an abstraction should be made with the `tailCall` instruction, as this lets the virtual machine optimise the execution (see below). It has the exact same arguments as the call instruction, only the name of the instruction differs: `tailCall(Callee Args)`.

`call` starts by creating a new frame on the stack with the currently running abstraction/code area, the program counter, and the Y registers. `tailCall` instead frees the Y registers. After that, the current abstraction and its code area are replaced by the callee and its code area.

```
move(y(4) x(0))
move(y(5) x(1))
move(y(6) x(2))
call(y(3) 3)
```

Figure 2.5: Opcodes for a call

A builtin is called with `callBuiltin(k(BI) Args)`, where `BI` must be a builtin (i.e. `{IsBuiltin BI} == true`).

`callBuiltin` always takes the builtin to call in a K register, but it is nonetheless specialised in different operations with 0 to 5 arguments or with an arbitrary number of arguments N. These are called `callBuiltinX` where X is one of `0 1 2 3 4 5 N`.

`return` cleans the Y registers and pops the top frame on the stack.

2.2.5 Records

Without being exhaustive, here is a reminder of the record concept in Oz.

A record is defined by `Label(F1:V1 F2:V2)` where the label and features (F_i) can be an integer, an atom or a variable, and the values (V_i) can any expression. Features can be left out, in which case they are implicitly assigned increasing values from 1.

At the virtual machine and compiler level, the arity of a record is made of its label and features. A tuple is a record whose features are consecutive integers starting from 1. In an abuse of language and only when no confusion is possible, we will sometimes refer to a feature's value simply as "feature". This simplifies the text without adding confusion, especially in the case of tuples where features are known to be numeric.

Records, tuples and cons are directly supported by the virtual machine, handling each of these particularly to ensure performance and efficiency. A record of arity Arity, with NumberOfFeatures features is unified with DestinationReg (an X, Y, G or K register) with this instruction:

```
createRecordUnify(k(Arity) NumberOfFeatures DestinationReg)
```

which has to be followed by instructions to fill the features with their respective values:

```
arrayFill(Reg)
```

where Reg is the register corresponding to the feature's value[3]. The arity is supported by the virtual machine and includes the label and features without their respective values.

There are two special cases of records that are handled specifically by the virtual machine. A cons is a record with label `'|'` and with only 2 features numbers 1 and 2 and is initialised by:

```
createConsUnify(DestinationReg)
```

[3] `arrayFill` might be renamed in `structFill` in the future

Again, this instruction has to be followed by `arrayFill` instructions, two in this particular case. A tuple is a record with all consecutive numeric features from 1. It is initialised with

```
createTupleUnify(k(Label) NumberOfFeatures DestinationReg)
```

and followed by `arrayFill` instructions.

Each of these instructions have a `Store` version (for example `createConsStore`) with the same arguments. This opcode stores the record created in the destination register which in this case must be an X or Y register, instead of unifying former with the entity created.

2.2.6 Procedures

A procedure is represented by what is called an abstraction (see Section 2.1.2), which is created and unified with the contents of register `DestinationReg` with

```
createAbstractionUnify(k(CodeArea) GlobalsCount  DestinationReg)
```

`CodeArea` is the assembled body of the procedure (see Section 3.1.6.6 for details). `GlobalsCount` identifies how many references to global variables this abstraction holds. It is the number of `arrayFill` instructions that will follow to initialise those global variables that the abstraction can access in G registers numbered from 0. The first `arrayFill` instruction will set the value of `g(0)`, the second the value of `g(1)`, ...

Figure 2.6 lists the Oz code we will analyse.

```
local
  A B P
in
  A=1           % in register X 0
  B=2           % in register X 1
  proc {P}      % in register Y 0
    {Show A+B}  % A and B are globals for P
  end           % access in g(0) and g(1) resp.
end
```

Figure 2.6: Oz code example for opcodes generation

As noted in the code comments, we suppose, the compiler having done its work of registers allocation, variables `A` and `B` are stored in registers `x(0)` and `x(1)` respectively, that the abstraction for procedure `P` is stored in register `y(0)` , and that `P` accesses the values of `A` and `B` via the registers `g(0)` and `g(1)` respectively. The body having been assembled and stored in the constant value `CodeArea`, this will lead to the code in Figure 2.7 to be passed to the assembler.

`createAbstractionStore` is used similarly to the records' `Store` instructions.

2.2.7 Pattern Matching

The instruction `patternMatch(Reg PatternMatchRecord)` supports the pattern matching, where `Reg` is the X, Y or G register containing the value to test,

```
createAbstractionUnify(k(CodeArea) 2  y(0))
arrayFill(x(0))
arrayFill(x(1))
```

Figure 2.7: Opcodes generated for example Oz code

and `PatternMatchRecord` is a record specifying the patterns and their respective jump address in case of a match. Figure 2.8 gives a simple Oz case instruction, and Figure 2.9 lists the generated opcodes. Section 3.1.6.8 covers pattern matching and the `patternMatch` instruction thoroughly.

```
case R
of l(1 2) then
   {Show 'l'}
[] rec(a b) then
   {Show 'rec'}
else
   {Show 'else'}
end
...
```

Figure 2.8: Example `case` instruction

```
              patternMatch(x(0) k((l(1 2)#6)#(rec(a b)#7)))
              branch(5)
lbl(6)        move(k(l) x(0))
              call(k(<P/1 Show>) 1)
              branch(3)
lbl(7)        move(k(rec) x(0))
              call(k(<P/1 Show>) 1)
              branch(3)
lbl(5)        move(k('else') x(0))
              call(k(<P/1 Show>) 1)
lbl(3)        ...
```

Figure 2.9: Opcodes for the example `case` instruction

2.2.8 Exceptions

There is an opcode to setup an exception handler and jump to a label:

```
setupExceptionHandler(DestLabel)
```

If the code executed from `DestLabel` raises an exception before the exception handler is removed with `popExceptionHandler`, the execution will jump to the opcode following the instruction `setupExceptionHandler`.

Figure 2.10 gives a simple example of Oz code handling exceptions, and Figure 2.11 gives an idea of corresponding opcodes. The opcodes corresponding

to `ExceptionHandlingOpCodes` and more detailed examples are given in Section 3.1.6.10.

```
try
   {Show 1}
catch E then
   {Show 'catched exception'}
end
```

Figure 2.10: Oz code handling exceptions

```
                setupExceptionHandler(1)
                ExceptionHandlingOpCodes
                branch(2)
lbl(1)          move(k(1) x(0))
                call(k(<P/1 Show>) 1)
                popExceptionHandler
lbl(2)          return
```

Figure 2.11: Opcodes handling exceptions

2.2.9 Skip

`skip` is a no-op opcode, that is actually dropped by the assembler. It is however available to ease the generation of the opcodes having some parts optional. Rather than testing if the opcodes sublist is empty before including it in the whole program's opcodes list, it is possible to generate it in every case: for the situation in which no operation is needed, a `skip` opcode will be issued. If the vm loop encounters a `skip` opcode, it does nothing and goes to the next opcode.

2.3 Compiler input

The compiler takes as input the AST in the form of Oz records [AST], and generates opcodes sent to the assembler. This section will present and describe the records received from the parser and their corresponding Oz form. Note that in de display of the ASTs, list can be represented by the square bracket notation and the `'|'` labeled records. For example, the list with elements `1` and `2` can be denoted `[1 2]` as well as `'|'(1 '|'(2 '|'(3 nil)))`.

2.3.1 Position in source code

Most records have the corresponding position in the source code available in their last feature, encoded in a record of the form:

`pos(File Linebegin Columnbegin Fileend Lineend Columnend)`

The position of each instruction has to trickle through all transformations so meaningful error messages can be given to the programmer in case of error. Most of the nodes in the AST include the position of their instruction in the source code.

For clarity and brevity, in the examples of AST included in this document, the positions will always be represented by the featureless record `pos` or a variable named `Pos`.

2.3.2 Literals

The basic data type described in this section are the simplest node found in the AST as they have no children and are always leafs of the AST. These are the basic data types with their corresponding tuples in the AST:

integers `fInt(Val Pos)`

floats `fFloat(Val Pos)`

atoms `fAtom(value position)`

`fAtom` are also used to place `unit`, `true` and `false` in the AST.

2.3.3 Identifiers

An identifier is denoted by a tuple `fVar(Ident Pos)` in the AST, where `Ident` is the text of the identifier represented as an atom.

2.3.4 Unification

A unification is found in a tuple `fEq(LHS RHS Pos)`. `A=3` results in `fEq(fVar(A pos) fInt(3 pos) pos)`.

2.3.5 Instructions sequence

A sequence of instructions is wrapped in fAnd records, the first feature being usually one instruction, the second feature being an fAnd if more than one instruction follows, or a single instruction. Here is an example of three unifications and the corresponding AST:

```
A=1                  fAnd( fEq (fVar (A pos) fInt (1 pos))
B=2                        fAnd(fEq (fVar(B pos)) fInt(2 pos)
C=3                             fEq(fVar(C pos) fInt(3 pos))))
```

2.3.6 Local

`local..in..end` are represented in the AST by tuples of the form

```
fLocal(Declarations Body Pos)
```

where `Declarations` and `Body` are both AST subtrees. Figure 2.12 shows Oz code declaring the three variables `A`, `B` and `C`, and bounding variable `A` with integer `1`. The AST corresponding to this Oz code is shown in Figure 2.13.

```
local
  A B C
in
  A=1
end
```

Figure 2.12: Declaration of 3 variables

```
fLocal(
       fAnd(
            fVar(A pos)
            fAnd(
                 fVar(B pos)
                 fVar(C pos)
                 pos)
            pos)
       fEq(
            fVar(A pos)
            fInt(1 pos) ))
```

Figure 2.13: AST for the code in Figure 2.12

2.3.7 Procedures

2.3.7.1 Definitions

Procedures are found in records of the form

```
fProc(Name Arguments Body Flags Pos)
```

where the `Arguments` and `Body` features are AST subtrees. Figure 2.14 and Figure 2.15 show a small procedure definition and its corresponding AST.

2.3.7.2 Calls

Procedure calls are found in records of the form `fApply(Proc Arguments Pos)`. Example:

```
{Compute 1 2 3}
```

will result in the AST portion:

```
fApply(
       [ fVar(Compute pos)
         fInt(1 pos)
         fInt(2 pos)
         fInt(3 pos)
       ])
```

2.3.8 Functions

Function definitions are found in the AST in `fFun` tuples, with the same features found in `fProc` records. Functions are simply procedures that implicitly return exactly one value. This similarity between functions and procedures can be

```
proc {P A1 A2}
   {Show A1}
   {Show A2}
end
```

Figure 2.14: Oz procedure definition

```
fProc(
   fVar('P' pos)
   [
     fVar('A1' pos)
     fVar('A2' pos)]
   fAnd(
      fApply(
         fVar('Show' pos)
         [fVar('A1' pos)]
         pos)
      fApply(
         fVar('Show' pos )
         [fVar('A2' pos)]
         pos))
   pos)
```

Figure 2.15: AST of the Oz procedure definition

seen in their similar AST for definitions and calls. It will be exploited later on to merge both forms of AST into one. However, only functions can be marked as lazy as in `fun lazy {F N} .. end`. This lazy marker is translated in a flag.

2.3.8.1 Definitions

Function definitions are found in records fFun(Name Arguments Body Flags Pos). The features are identical to the procedure definition.

2.3.8.2 Calls

The calls of functions are identical to the call of procedures as they are found in fApply.

2.3.9 Operators

Operators are parsed as records of the form

```
fOpApply(Operator Args Pos)
```

whose first feature is the operator to apply, its second feature being the list of operands. It also has a position record as third feature. For example `5+2` is parsed as displayed in Figure 2.16.

```
fOpApply(
   '+'
   [
     fInt(5 pos)
     fInt(2 pos )
   ]
   pos
)
```

Figure 2.16: AST for `5+2`

2.3.10 Nesting marker $

In Oz, some statements can be transformed in expressions by the use of the `$` marker. This marker is found in the AST as a tuple `fDollar()` with a unique feature: its position. The statements that can be transformed in expressions are

- Function, Procedure, Functor and Class declarations.
  For example `X=fun{$ A B} .. end`
- A call to a procedure having the nested procedure in one of the pattern positions of its arguments.

The pattern positions `PP` of an argument being the union of

- the argument itself
- if it is a record of the form `l(f1:v1 ... fn:vn)`, $PP(v1) \cup ... \cup PP(vn)$.

2.3.11 Cells

2.3.11.1 Assignation

A new value can be stored in a cell with this Oz code: `Cell:=Val`. This code will result in this tuple in the AST: `fColonEquals(Cell Val Pos)`.

2.3.11.2 Read

The value stored in a cell can be access with the `@` operator, as is `@Cell`, which results in this tuple in the AST: `fAt(Cell Pos)`

2.3.12 Records

Records are present in the AST as tuple with label `fRecord`. Its label is the first feature, and the second feature is the list of feature-value pairs. If no explicit feature was specified, the entry in the list is simply the value. If an explicit feature was specified, the entry in the list is a tuple `fColon`, with the first feature being the feature and second feature being the value. All this will become much clearer with the following examples.

Figure 2.17 displays the AST corresponding to the record `rec(f1:v1 f2:v2)`. We see that the label (rec) is an atom located in an `fAtom` tuple. The pairs of features and their respective values are wrapped in fColon.

```
fRecord(
   fAtom(rec pos)
   [
    fColon(
      fAtom(f1 pos)
      fAtom(v1 pos))
    fColon(
      fAtom(f2 pos)
      fAtom(v2 pos))])
```

Figure 2.17: Record AST

When the feature is not explicitly given, the item in the list of features and their respective values is simply the value. Figure 2.18 is the AST of `rec(v1 f2:V2)`. The first item in the list is simply an atom, the value of the first feature. Note that the value of the feature f2 is a variable in this case.

```
fRecord(
   fAtom(rec pos)
   [
    fAtom(v1 pos)
    fColon(
       fAtom(f2 pos)
       fVar(V2 pos))])
```

Figure 2.18: AST for record with implicit feature

2.3.13 Wildcards

Oz syntax allows to put a wildcard `_` in the location of a value we want to ignore, for example in calls and records pattern matching. These markers are present in the AST in the form of a tuple `fWildcard(Pos)`. Ignoring the value returned by a function is thus written `_={F N}`. This allows the user to not declare variables he wouldn't use anyway, which would in addition raise a warning about a variable used only once...

Pattern matching is covered extensively in Section 2.3.18.

2.3.14 Threads

Thread instructions are present in the AST in the form of tuples with label `fThread`, and with two features: the body of the thread, and its position in the source code:

```
thread Body end
```

results in

```
fThread(Body pos)
```

2.3.15 Locks

Oz enables the programmer to protect a critical section with a lock previously created with `L={NewLock}`:

```
lock L then
   % critical section
end
```

This code results in a tuple `fLockThen` in the AST, with 3 features: the lock L, the body of the critical section, and the position in the code: `fLockThen(L Body Pos)`.

2.3.16 If then else

The `if..then..else` construct is put in a tuple:

```
fBoolCase( Condition ThenCode ElseCode)
```

The else part is optional in Oz. In this case, ElseCode is a tuple `fNoElse(pos)` in the AST.

2.3.17 Short-circuit boolean combinators

The boolean conjunction operation in Oz is represented by the keyword `andthen`. Such a conjunction is present in the AST in the form of a tuple

```
fAndThen(First Second)
```

where first and second must be expressions with a boolean value. The boolean disjunction operation is done using the keyword `orelse`, present in the AST in the form of a tuple `fOrElse(First Second)`.

2.3.18 Case Instruction and Pattern Matching

The Case instruction in Oz is of the following form where optional parts are enclosed in double square brackets ([[ ]]).

```
case Value                              % _
of Pattern1 [[andthen Guards1]] then %  \____ This is a clause
  Code1                                 % _/
[] Pattern2 [[andthen Guards2]] then
  Code2
..
[[else
  ElseCode]]
end
```

This ends up in the AST in a tuple of the form

```
fCase(Value Clauses ElseCode pos)
```

`Clauses` is a list of tuples each representing one clause and containing the clause' pattern, guards and code. We will analyse these further below. The else part of the instruction is optional. If no else is provided, then ElseCode in the AST is a tuple `fNoElse(pos)`.

A clause with no guards is present in the AST in the form of a tuple `fCaseClause(Pattern Body)`. `Pattern` can be a constant value (integer, string,

record, ...), present in the AST as described earlier. It can also be a record where a feature's value is not a constant but a variable (in a `fVar(Name pos)` record as described in section 2.3.3). In case of a match of the constant feature values in the pattern, this declares said variable and assigns the value of the corresponding feature in the value the pattern is tested against. Here is an example:

```
R=lab(a b c d)
case R
of lab(a B c d) then
  {Show B}  % B's value is b
end
```

Features of a record in a pattern can also have the wildcard `_` as value, represented in the AST as described in section 2.3.13.

The pattern can also be or contain an open record specifying only a subset of features which should match, as in the following example:

```
R=lab(a b c d)
case R
of lab(3:c 4:_ ...) then
  % Code for record matching pattern
end
```

The syntax for an open record pattern is the same as for a record pattern, except that subsequent features that should be ignored are replaced by `....` Also in the AST an open record is very similar to a record. Only the label changes to `fOpenRecord`, and only the specified features are present in the AST, the `...` being absent from the AST. Here is an example of an open record: `lab(a b ...)` and its AST representation:

```
fOpenRecord(
   fConst(lab pos)
   [
    fColon(
       fConst(1 pos)
       fConst(a pos))
    fColon(
       fConst(2 pos)
       fConst(b pos))])
```

Records and open records can of course be nested.

A clause with guards is present in the AST in the form of a tuple

```
fSideCondition(Pattern Decls Guards Pos)
```

where `Pattern` is as above, and `Guards` is simply the AST of the code of the guard (without the `andthen` introducing the guard). `Guards` must be an expression with a boolean value, possibly using the short-circuit boolean combinators of section 2.3.17.

`Decls` is the declarations introduced by the guards, as the variable `X` in the code of Figure 2.19. When there are no declarations, the value is `fSkip(pos)`

Variables can also be used in patterns for their value, and not as a new capture declaration. This is done by prefixing the variable with an exclamation mark. Here is an example:

```
case Foo
of bar(F) andthen X = F+4 in X > 0 then
   {Show X}
end
```

Figure 2.19: Guards declarations

```
A=a
case V
of rec(A b c) then
   % matches rec(_ b c), i.e. all records with label rec, second feature
   % value b, third value c, with A being a new capture variable
[] rec(!A b c) then
   % only matches rect(a b c)
end
```

An escaped variable ends up in the AST as wrapped in an `fEscape` tuple. `!A` appears in the AST as `fEscape( fVar(A pos) pos)`.

Oz pattern matching also support what is called pattern conjunction, enabling to capture the value of a subexpression of the pattern as illustrated in this example:

```
case V
of rec(I=inner(a b ...)) then
...
end
```

In the clause' code, the variable `I` will have the value of the inner record. These pattern conjunctions are present in the AST in `fEq` tuples just like unification instructions. Note that just like unifications (and unlike `local..in..end`), the variable implicitly declared can be at the left as well as at the right side of the equal sign. So the pattern `rec(V=inner(a b ...))` is equivalent to `rec(inner(a b ...)=V)` And also like in unification, the capturing side can be a pattern, it doesn't have to be a single variable.

```
case {GetToken}
of unit then
   skip
elseof S then
   {Handle S}
end
```

Figure 2.20: `elseof` construct

2.3.19 Classes

Analysing classes is best illustrated by an example of Oz code:

```
class C
   meth init skip end
   meth hello(A1)
      {Show 'hello'}
      {Show A1}
   end
end
```

which results in this AST:

```
fClass(
   fVar('C' pos )
   nil
   [
     fMeth(
        fAtom(init pos )
        fSkip(pos )
        pos)
     fMeth(
        fRecord(
           fAtom(hello pos )
           [
             fMethArg(
                fVar('A1' pos )
                fNoDefault)
           ])
        fAnd(
           fApply(
              fVar('Show' pos )
              [
                 fAtom(hello pos )
              ]
              pos)
           fApply(
              fVar('Show' pos )
              [
                 fVar('A1' pos )
              ]
              pos))
        pos)
    ]
   pos)
```

Class definitions are found in `fClass(ClassVar SpecsList MethodsList Pos)`. A method is found in a tuple `fMeth(Signature Body Pos)`. For methods without argument, `Signature` is simply the method name. In the example, this is illustrated by the `init` method. For methods with arguments, `Signature` is the AST of a record whose label is the method name, and the values in the record are the method arguments found in `fMethArg(Var DefaultValue)`, where `DefaultValue` is `fNoDefault` if none was provided. This is illustrated by the method `hello(A1)`.

The class of this example had no feature, attribute, parent or property defined, hence the empty list as value of the second feature. Both attribute and features are placed in this list. Attributes are placed in `fAttr(AttrsList)` tuples and features are placed in `fFeat(FeatsList)` tuples. Both these tuples have one feature whose value is a list. Each attribute and feature corresponds to one item in its respective list. If the attribute or feature has a default value, its corresponding item is a record with label `'#'` and two values: the first is the name of the attribute or feature, the second is the default value. If no default value is defined, the item in the list is simply the name of the attribute or the feature. Let's look at an example. A class with the following attributes and feature definition

```
attr
   count:0
   state
feat
   type:repeater
```

will have its list of attributes and features looking like this:

```
[
   fAttr(
      [ #( fAtom(count pos)
           fInt(0 pos ))
        fAtom(state pos ) ]
      pos)
   fFeat(
      [ #(
          fAtom(type pos )
          fAtom(repeater pos )) ]
      pos)
]
```

The `count` attribute is placed in a '#' record, but not the `state` attribute. All attributes are placed in one `fAttr` tuple, and all features in one `fFeat` tuple.

This list also contains more information: the parents of the class, as well as the properties of the class.

The parents are located in `fFrom(ParentsList)`. Here is an example from a class inheriting from classes `A` and `B`. In Oz, this class will be defined by

```
class C from A B
  ..
end
```

and this will result in this element in the list

```
fFrom(
   [ fVar('A' pos )
     fVar('B' pos )]
 pos)
```

The properties of a class are also present in that list, under `fProp` tuples. A class having the locking property would have this `fProp` tuple:

```
fProp(
    [fAtom(locking pos)]
    pos)
```

When a class has the `locking` property, a lock is implicitly created, that can be used in a method with the `lock` statement without specifying the lock object, as this:

```
meth update(A)
   lock
      ..
   end
end
```

This `lock` instruction is put in the AST as a node `fLock(Body Pos)`.

Assignation to attributes can be done in two ways. The first has the same syntax as cell updates, described in Section 2.3.11:

```
my_attr:=NewVal
```

and is found in the AST just as a cell update operation under a node `fColonEquals`.

The second uses the syntax `<-`, as in:

```
my_attr<-NewVal
```

and is found in the AST under a node `fAssign(LHS RHS pos)`.

The method head can also be captured and made available to its body,as in

```
meth echo(First ...)=H
   Body
end
```

At run time, the `H` variable will give access to the message sent to the object, with the actual arguments list.

```
fMeth(
   fEq(
      fOpenRecord(
         fAtom(echo pos )
         [ fMethArg(
               fVar('First' pos )
               fNoDefault)])
      fVar('H' pos )
      pos)
   BodyAST
   Pos)
```

Method can also be defined as private by using a variable name as their label:

```
meth A(V)
   Body
end
```

resulting in the AST in Figure 2.21.

The method `A` will be bound to a new name, only visible in the class definition's scope (see Section 3.1.2.5 for details).

Method can also have dynamic labels assigned, by escaping the variable holding the label to use:

```
meth !A(V)
   Body
end
```

```
fMeth(
   fRecord(
      fVar('A' pos )
      [ fMethArg(
           fVar('V' pos )
           fNoDefault)]
   BodyAST
   pos)
```

Figure 2.21: Private method AST

The AST for a dynamic label assigned to a method is:

```
fMeth(
   fRecord(
      fEscape(
         fVar('A' pos)
         pos)
      [ fMethArg(
            fVar('V' pos)
            fNoDefault) ]
   BodyAST
   pos)
```

2.3.20 Loops

There are three forms of the `for..in..do..end` statement, working respectively on list, ranges, and C-like for loops conditions.

2.3.20.1 Iterating over lists

The Oz syntax is best illustrated by an example:

```
for Var in L do {Show Var} end
```

`Var` is bound to each element of L in turn, and the code executed. Hence this code will show all elements of the list L.

Multiple list can be iterated:

```
for I in L J in L2 do {Show I#J} end
```

but note that the iteration goes over the two zipped lists , and the loop ends once the end of one of the list is reached. The number of times the body of the loop is called is thus the number of elements of the smallest list.

The AST of the first `for` example is displayed in Figure 2.22.

The first value in the `fFor` tuple is the list of patterns, one for each list. Each item of this list is a `forPattern` tuple, with its first value being the variable that will be available in the loop, and the second value an `fGeneratorList` tuple referencing the list over which to iterate. The second value in the `fFor` tuple is the code to be executed for each element of the list.

```
fFOR(
   [forPattern(
        fVar('Var' pos )
        forGeneratorList(
           fVar('L' pos ))) ]
   fApply(
      fVar('Show' pos )
      [fVar('Var' pos)]
      pos)
   pos)
```

Figure 2.22: For loop iterating over list `L`

2.3.20.2 Iterating over ranges

Here is an example of a `for` loop iterating over integers from 1 to 5 with a step of 2, resulting the execution of the body of the loop for integers 1 3 and 5:

```
for I in 1..5;2 do {Show I} end
```

```
fFOR(
   |(
      forPattern(
         fVar('I' pos )
         forGeneratorInt(
            fInt(1 pos )
            fInt(5 pos )
            fInt(2 pos )))
      nil)
   fApply(
      fVar('Show' pos )
      |(
         fVar('I' pos )
         nil)
      pos)
   pos)
```

The AST is very similar to the first loop form, only the `fGeneratorList` tuple is replaced by a `forGeneratorInt` tuple holding the specification of the range and step to use.

2.3.20.3 C-style for loops

Iterating over integers from 1 to 5 with a step of 2 can also be written in the C-style for loop:

```
for I in  1;I<4;I+1 do  {Show I} end
```

It takes a generator composed of 3 expressions: `E1;E2;E3` where `E1` is the start value, `E2` is the test condition, the looping continuing as long at it evaluates to true, and `E3` is the next value's expression. It is found in the AST illustrated in Figure 2.23.

The notable difference is the generator tuple now being `forGeneratorC` holding the three expressions.

```
fFOR(
   [ forPattern(
        fVar('I' pos )
        forGeneratorC(
           fInt(1 pos )
           fOpApply(
              '<'
              [ fVar('I' pos )
                fInt(4 pos ) ]
              pos)
           fOpApply(
              '+'
              [ fVar('I' pos )
                fInt(1 pos ) ]
              pos))) ]
   fApply(
      fVar('Show' pos )
      [ fVar('I' pos ) ]
      pos)
   pos)
```

Figure 2.23: C-like loop

2.3.21 Exceptions

An exception can be raised with the instruction `raise E end`. A raise instruction is represented in the AST by a tuple `fRaise(E pos)`.

Code possibly raising exceptions can be wrapped in a try-catch-finally instruction of the form illustrated in figure 2.24.

```
try Code
catch
   Pat1 then
   Code1
[] Pat2 then
   Code2
finally
   FinalCode
end
```

Figure 2.24: Try-cat-finally structure

We can see that the catch part specifies multiple clauses each with a pattern against which the exception raised will be tested, and code which will be executed if the pattern effectively matched. `finally` introduces code that will be executed in all cases after `Code` and the possible matching catch clause.

It results in the AST of Figure 2.25.

which exactly mirrors the Oz code: `fTry` holds the AST of the body, which is code possibly throwing an exception, the `fCatch` tuple containing the list of clauses, and the `finally` code. The `finally` may be left out of the Oz code, in

```
fTry(
   Body
   fCatch(
      [ fCaseClause(
           Pat1
           Code1)
        fCaseClause(
           Pat2
           Code2) ]
      pos)
   FinalCode
   pos)
```

Figure 2.25: Try-catch-finally AST

which case `FinalCode` is the tuple `fNoFinally`. Clauses of a catch are present in the AST in the same form as clauses of a case instruction (see Section 2.3.18), i.e. a `fCaseClause` tuple holding the pattern and the code to execute if the pattern matches. This similarity will be exploited by the compiler, notably in the namer, which is the next transformation to be applied to an `fTry` node (see Section 3.1.2.6).

2.3.22 Arrays

A special notation is available for assigning value to array elements, as to mimic the matrix notation. As such, the code `A.I:=V` will assign `V` to the `I`th element in the array `A`. This results in node `fDotAssign` in the AST. The node generated by the parser is

```
fDotAssign(fOpApply('.' [A I] Pos1)
           V
           Pos2)
```

The first transformation applied to these nodes takes place in the desugar step, described in Section 3.1.3.16.

2.3.23 Functors

Modular applications can be developed in Oz by using functors. A functor computes a module, taking modules as input and producing a new module as output. A module groups together related operations, and consists of an interface and an implementation. The interface publishes entry points to the module implementation, the rest of the implementation being unreachable from the outside. A functor can `import` other modules, `export` entry points to its implementation, and `define` its implementation, as illustrated by the code in Figure 2.26.

Two important steps are distinguished in the life of a functor: it is first evaluated, and then it is applied. These two steps bring a distinction between compiled and computed functors [Func]

```
functor
import
   DumpAST at '../lib/DumpAST.ozf'
export
   PrettyPrint
define
   proc {PrettyPrint AST}
      {DumpAST.dumpAST AST _}
   end
end
```

Figure 2.26: Compiled functor

A computed functor has its name coming from the fact that some of its code is evaluated at definition time, resulting in a functor that has been computed, in opposition to compiled functors.

The code executed when evaluating computed functors can for example compute a data structure to be carried with the computed functor.

Computed functors are supported in Oz with `require` and `prepare`, which define code parts to be evaluated when building the functor. This is illustrated by the code in Figure 2.27, coming from the online Oz documentation [Func].

```
functor
import
   DB Form            % User defined
   System Application % System
require
   BuildSampleFlights
prepare
   Flights = {BuildSampleFlights}
define
   %% Enter some flights
   {ForAll Flights DB.add}
   ...
end
```

Figure 2.27: Computed functor

The key point in this functor definition is the presence of the `require` and `prepare` sections. When this functor is evaluated, the `require` and `prepare` are evaluated. Hence the computed functor contains the sample flights list. This means that the code will have access sample flights that were selected when the functor was evaluated (built), and not when it was applied (imported).

Functors appear in the AST in `fFunctor`:

```
fFunctor(Id ImportRequirePrepareDefineExportList Pos)
```

The first value, `Id`, is the identifier of the functor, either `fDollar` for anonymous functors, or `fVar`. The second value is a list containing all `import`, `require`, `prepare`, `define` and `export` information.

We will illustrate the explanations by using the example code in Figure 2.28.

```
functor
export
   echo:Echo
import
   Helpers(print:Print) at '../lib/Helpers.ozf'
prepare
   Test=1000
define
   proc {Echo S}
      {PrivateEcho S}
   end
   proc {PrivateEcho S}
      {Print Test}
      {Print S}
   end
end
```

Figure 2.28: Example functor

The `import` information is present in a tuple `fImport` holding a list of tuples `fImportItem(Id Aliases Location)` as illustrated in Figure 2.29. The `Id` is simply the variable defined for the `import`. `Aliases` is a list of pairs `'#'(Var Key)` binding `Var` to the function exported by the imported functor under the key `Key`. `Location` is either `fNoImportAt` if no location was given, or `fImportAt(Path)` specifying the relative URL of the imported functor. `require` is placed in a tuple `fRequire` of exactly the same form, and is not illustrated here.

```
fImport(
   |(
      fImportItem(
         fVar('Helpers' pos )
         |(
            #(
               fVar('Print' pos )
               fAtom(print pos ))
            nil
         ) fImportAt(
            fAtom('../lib/Helpers.ozf' pos )))
      nil)
   pos)
```

Figure 2.29: Import AST example

The `prepare` section is placed in a tuple `fPrepare(Decls Stats Pos)`, holding the `prepare`'s declarations and statements, as illustrated in Figure 2.30. Our example contained only declarations, and the statement part is a `skip` instruction. `define` is placed in a tuple `fDefine` of exactly the same form, and is not illustrated here.

```
|(
 fPrepare(
    fEq(
       fVar('Test' pos )
       fInt(1000 pos )
       pos)
    fSkip(pos )
    pos
 )
```

Figure 2.30: Prepare AST example

`export` is a tuple `fRecord( [fExportItem(fColon(Key Var))...])`, specifying `Key` as the key in the module's interface to access `Var`, as illustrated in Figure 2.31.

```
fExport(
   |(
      fExportItem(
         fColon(
            fAtom(echo pos )
            fVar('Echo' pos )))
      nil)
   pos)
```

Figure 2.31: Export AST example

Chapter 3

Compiler

3.1 Architecture

The AST received from the parser goes through several transformations called passes. Each pass has its specific purpose and applies a specific transformation to the AST. The compiler's passes are illustrated in Figure 3.1

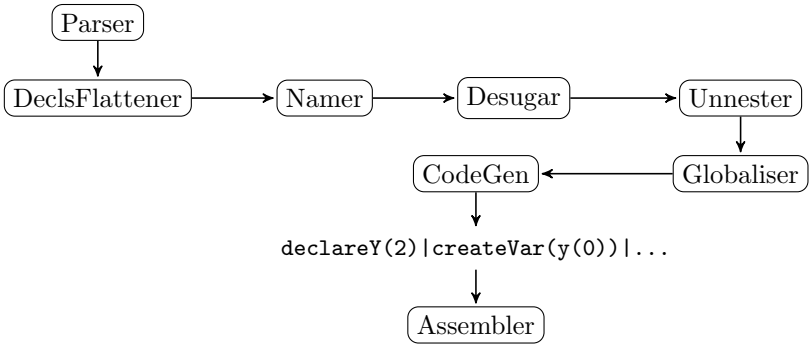

Figure 3.1: Compiler phases

These transformations are implemented by functions taking as only argument the AST, and they return the transformed AST. These functions all have the same structure, as displayed in Figure 3.2. Having the same structure enables the use of the function `DefaultPass` listed in Figure 3.3, which calls the function passed as second arguments recursively on the AST with `Params` as last argument, but returns the root of the AST unchanged. This facilitates the recursive calls for default behaviour on elements not modified by the transformation.

Params is a record with label `params`, and features are added as necessary. For example, this `Params` argument is used by the `Globaliser` described in Section 3.1.5. All passes take a `params` record as last argument, even if some do not use it. This early design decision proved very beneficial afterwards.

Although all passes work exclusively on the AST, the transformations described in later sections can and will often be illustrated more clearly and concisely by giving the Oz code corresponding to the ASTs before and after the transformation.

```
fun {Transform AST}
   fun {TransformInt AST Params}
      case AST
      of ... then
      else
         {DefaultPass AST TransformInt Params}
      end
   end
   InitialParams = ...
in
   {TransformInt AST InitialParams}
end
```

Figure 3.2: Structure of a transformation function

```
fun {DefaultPass AST F Params}
   % beware of the order. a record is also a list!
   if {List.is AST} then
      {List.map AST fun {$ I} {F I Params} end}
   elseif {Record.is AST} then
      case AST
      of pos(_ _ _ _ _ _) then
         % Do not go down into position records
         AST
      else
         {Record.map AST fun {$ I} {F I Params} end}
      end
   else
      AST
   end
end
```

Figure 3.3: DefaultPass function

3.1.1 Declarations Flattener

The Oz language accepts complex instructions in the declaration part of

```
local Decls in Body end
```

It is thus necessary to identify the variables declared by each instruction present in the `Decls`. For example, in unifications, only the left hand side variable gets declared, and only `B` is implicitly declared in the following code snippet:

```
local
   B=2*A
in
   ...
end
```

Fortunately, this has been clearly formalised by the Oz implementers and published on the web [BaseLang]. This information has been included in Appendix

A. The namer simply applies these rules to move all code to the `Body` beginning, keeping only simple variable declarations in `Decls`.

3.1.2 Namer

The namer basically replaces all occurrences of a variable's `fVar` by `fSym` tuples holding a symbol instance representing said variable. All uses of a variables are replaced by the same symbol instance. This symbol holds information about the variable, that will be completed by subsequent compiler passes, such as the type of the symbol, its allocated register,... and is documented in Appendix C.

In declarations, the namer creates a new symbol for the declared variables, in other code, it looks at already defined symbols and replaces `fVar`s by a `fSym` tuple containing the existing symbol for that variable. This works because a variable can only be in the subtree whose root is the variable declaration's instruction (be it a `local Decls in Body end` declaring variables in `Decls`, a `proc {Id P1 P2}` declaring its formal parameters, ...) . The namer can thus be sure to find a symbol for variables it encounters in the AST, or it is an error.

The `Namer` also has some work to do on patterns and the variables it implicitly declares. As a consequence, it also handles and transforms the AST for functions and procedures, which implicitly declare their formal parameters.

The set of maps from a variable name to its corresponding symbol is called environment, and is maintained by the namer.

3.1.2.1 Declarations

Variables can be declared in different ways and instructions. One such instruction is of course the explicit declaration instruction `local Decls in Body end`, present in the AST in the form of `fLocal` tuples introduced in section 2.3.6. At this stage, `Decls` only contains simple variables declaration, thanks to the declaration flattener (see Section 3.1.1). `fLocal` introduces a new environment, in which the variables declared in its first feature are added to the environment of the code present in its second feature, possibly erasing mappings for variables with the same name coming from the parent environment.

It is these manipulations of the environments that make that the code in Figure 3.4 does not throws an error of impossible unification of A with 6. Indeed, the inner variable A gets another symbol assigned than the outer one, and replaces it in the environment of the inner instructions. Thus there is no ambiguity in the `Show` instruction as to which its argument is, and it effectively shows 6.

Another kind of declaration is found in the definition of procedures and functions. Indeed, `fProc` (Section 2.3.7) and `fFun` (Section 2.3.8) tuples also implicitly declare the variables for their formal parameters.

Still another kind of declaration happens in pattern matching (Section 2.3.18), when a record present in the pattern has a feature that is a variable. This case is analysed specifically in Section 3.1.2.3.

These variables get implicitly declared and added to the environment in which not only the clause' body but also its guards will be executed.

```
local
   A=5
in
   local
      A=6
   in
      {Show A}
   end
end
```

Figure 3.4: Nested locals handled by Namer

3.1.2.2 Non-declaration code

In code not declaring variables, the namer's transformation is much simpler: it will simply replace all occurrences of `fVar` tuples by a `fSym` tuple having the corresponding symbol as first feature.

3.1.2.3 Case Pattern Matching

Let's note `PatternX(Var1,Var2,..)` the Xth pattern tested and implicitly declaring variables `Var1`,`Var2`,..., and let's use the same convention for code fragments `CodeX(Var1,Var2,..)` and guards `GuardsX(Var1,Var2,..)` using those implicitly declared variables.

The transformation of the AST corresponding this Oz code:

```
case Val
of Pattern1(A,B) andthen Guards1(A) then
   Code1(A,B)
[] Pattern2(B,C) andthen Guards2(C) then
   Code2(B,C)
else
   ElseCode
end
```

results in an AST corresponding to this Oz code:

```
local
  A1 B1 B2 C2
in
   case Val
   of Pattern1(A1,B1) andthen Guards1(A1) then
      Code1(A1,B1)
   [] Pattern2(B2,C2) andthen Guards2(C2) then
      Code2(B2,C2)
   else
      ElseCode
   end
end
```

Here is the way how it is achieved. The compiler traverses each clause in turn, and collects all symbols introduced during this operation. It creates a new environment for this clause, which is updated with captures done in the pattern. After that, the namer traverses the ASTs of the Guards and the

Code using this updated environment, and restores the original environment, on which the next clause will build its own environment. This shows clearly that although all capture variables are technically reachable from all clauses' bodies, it will never be the case as the environment on which a clause builds its own is the environment of the case instruction, not the environment updated by the previous clause.

In addition, the `fSym` tuple corresponding to these capture variables are themselves stored in an unforgeable secure structure (as described in [VRH04] Section 3.7.5) which is itself wrapped in an `fConst`. The reason of this manipulation is to prevent manipulations of those symbols by later compiler passes. Now these will be handled as constants in all subsequent compiler passes. Another consequence of this is that the records implicitly declaring variables in a pattern are seen as constant records, all features being constants. And these are treated as such, notably in the desugaring of records (section 3.1.3.6).

To ease the work of future passes and improve readability of their code, the namer also replaces `fEq` tuples of pattern conjunctions by `fPatMatConjunction` tuples.

Patterns with guards require some additional work. These are present in the AST as

```
fSideCondition(Pattern Decls Guards Pos)
```

First, a new symbol is introduced and unified with the `Guards` expression. The record in the AST is replaced by

```
fNamedSideCondition(Pattern Decls GuardsStatement GuardSymbol Pos)
```

where `GuardStatement` is the unification of the newly introduced symbol `GuardSymbol`. The `GuardSymbol` is declared together with capture variables. This transformation cannot be completely represented in Oz code, notably the transformation of the guards in statement. It will become clear when looking at the `CodeGen` why this transformation is done like this (the description of the CodeGen code Section 3.1.6.11 has more details on page 81).

Escaped variables are transformed by the namer, which introduces a capture symbol and then validates in a new guard that it has the value of the escaped variable. A pattern with an escaped variables, like `rec(!Var)` is transformed in `rev(Capt) and then Capt==Var` Although this transformation might be seen as a `Desugar` transformation, it is done in the `Namer` because it already traverses and transforms the patterns, and all code infrastructure was already in place. Putting this transformation in the `Desugar` pass would have introduced code duplication.

3.1.2.4 Pattern Arguments

Functions and procedures in Oz can be defined with pattern arguments, which destructures the argument and can shorten significantly the code:

```
fun {GetName person(name:Name ...)}
   Name
end

Name={GetName person(name:'Turing' firstname:'Alan')}
```

The AST of this function's definition is actually transformed in an AST corresponding to the code listed in Figure 3.5.

```
fun {GetName Arg }
   case Arg
   of person(name:Name ...) then
      Name
   end
end
```

Figure 3.5: Pattern argument transformation result

The namer thus replaces pattern variables by a synthetic symbol, and wraps the body of the function in case instructions (one level of case nesting for each pattern argument).

3.1.2.5 Classes

The work of the namer on classes is in most cases straight-forward. It goes through the tuple `fClass(Id Specs Methods Pos)` and names variables as one would expect: method arguments are named as declarations i.e. it adds mappings to the environment which are then used when traversing the body of the method. It also goes through the attributes, features, properties and parents.

Difficulties arise though when methods have dynamic names, are private and/or capture their head.

For method with a dynamic name, the namer simply has to use the binding for the escaped variable in the current environment.

Private methods have their label defined as a variable. The namer will transform this by declaring a new symbol for the variable and bind it to a new name. The transformation can be illustrated in Oz code, transforming

```
class C
   meth init skip end
   meth A()
      Body
   end
end
```

in

```
local
   A
in
   A={NewName}
   class C
      meth init skip end
      meth !A()
         Body
      end
   end
```

Note also that the variable being the label of the method can have the same name as the class. Inside the class the variable will reference the method, and

not the class, as illustrated in Figure 3.6. The private methods' variables are available inside the class, overriding the mapping to the class `C`.

```
class C
   meth init skip end
   meth C(M)
      {Show M}
   end
   meth whisper(M)
      {self C(M)}
   end
end
```

Figure 3.6: Private method overriding the mapping of C

We can now look at the situation when the method head is captured (see Section 2.3.19 page 22 for the AST form). Figure 3.7 lists code illustrating the specific points requiring attention.

```
class C
   meth init skip end
   meth echo(F ...)=C
      {Show C}
   end
   meth whisper(F ...)=C
      {Show F}
   end
   meth clone()
     {New C init}
   end
end
```

Figure 3.7: Class definition

These attention points are:

- the same variable name can be used by multiple methods to store their respective method heads
- the variable holding the method head has to be available only to the method defining it
- the variable can have the same name as the class in which the method is defined (`C` in method `clone` references the class `C`)

Again, when the namer enters a method, it defines a new environment.

Having covered all special behaviour to support, we can now look at how the namer works to respect these constraints.

We have seen that the namer has to be able to declare and initialise new variables which will be available inside the class only. As a consequence, the namer will maintain a list of symbols to be declared with their respective initialisation code.

The namer starts by naming the class.

It then creates a new environment, and names all method labels (and only the labels, not the arguments, not the variable capturing the method head). Variables named in this step will be available to all methods of the class. They correspond to labels of private methods, and their initialisation code is simply unifying them with a new name.

It will then go through all methods again to name their respective arguments, body and possible method head capture variable. For each method it will take these actions in order

1. create new environment
2. name the method head capture variable as declaration if it was defined
3. name the method's arguments as declaration
4. name the body
5. restore the previous environment

It then wraps the class definition in a `local..in..end` declaring the variables to be declared collected while traversing the class (these are variables that were encountered as method label), and prepend the class definition by the initialisation code collected:

3.1.2.6 Exceptions

The AST received by the namer for a try instruction is:

```
fTry(
   Body
   fCatch(
      |(
         fCaseClause(
            Pat1
            Code1)
         |(
            fCaseClause(
               Pat2
               Code2)
            nil))
      pos)
   FinalCode
   pos)
```

The namer will traverse `Code` and `Finally` and name variables based on existing mappings present in the environment. The patterns of a catch clause is handled exactly (in the sense that it is the same compiler code that is called) as the pattern of a case clause (see Section 3.1.2.3). A new environment is created for each clause, to which the pattern might add bindings, which are then available when naming the code of the clause. New symbols are possibly introduced by the patterns, which then have to be declared in a `local..in..end` instruction wrapping the named `try` instruction. This construction is very similar to the one presented in Section 3.1.2.3.

3.1.2.7 Functors

The work done by the namer on `fFunctor(Id SpecsList Pos)` nodes is trivial, but the order of naming the items of the `SpecsList` is important. For example, `import`ed variables have to be named before traversing the `define` block, or else these wouldn't be available to that code!

It is possible to sketch the scope of variables introduced by a functor with the code in Figure 3.8. The variables declared by `require` are available to all other parts of the functor, which is not the case of the variables defined by `prepare`, which are not available in the `require` statements.

```
local
   require_decls
in
   require_stats
   local
      prepare_decls
   in
      prepare_stats
      local
         import_decls
      in
         import_stats
         local
            define_decls
         in
            define_stats
            exports
         end
      end
   end
end
```

Figure 3.8: Functor variables scopes

One other small change introduced by the namer is the creation of default export key, changing

```
export
   DumpAST
```

in

```
export
   dumpAst:DumpAST
```

3.1.2.8 For loops

Loops over a list are transformed in a call to `ForAll` which is then recursively named. Note that the behaviour of a for loop over multiple lists is not correct with this implementation and as a consequence the test number 133 of the test suite does not currently pass. The correction of this code is mentioned in the "Future work" section on page 91.

Loops over an integer range are first transformed in a call to `for` and then recursively named.

C-like loops are currently not supported.

3.1.2.9 Compiler Code Notes

The namer is one function unimaginatively named `Namer` taking as argument an AST, and returning the resulting AST. The following local functions are defined: `NamerForDecls`,`NamerForBody`,`NamerForCaptures`.

`NamerForBody` calls `NamerForDecls` when a function or procedure is defined, and calls `NamerForCaptures` for patterns in `fCaseClause`s. Figure 3.9 lists the code handling a definition in non-declaration part of a function with no pattern argument:

```
%---------------------------------
[] fFun(Name Args Body Flags Pos) then
%---------------------------------
   Res
in
   {Params.env backup()}
   Res=fFun(
      % The function's variable has to be declared explicitely
      % in the declaration part.
      % That's why we call NamerForBody on the Name
      {NamerForBody Name Params}
      % Formal parameters are declarations, that's why we
      % call NameForDecls
      {List.map Args fun {$ I} {NamerForDecls I Params} end }
      {NamerForBody Body Params}
      Flags
      Pos
   )
   {Params.env restore()}
   Res
```

Figure 3.9: Namer transformation of fFun nodes

The environment, mapping variable names to their respective symbol, is stored in the object available through `Params.env`. This object's `backup` method will backup the current environment and create a new one with the exact same bindings. The function creates a new environment based on its parent environment, which is backed up to be restored when done. The name of the function is passed to `NamerForBody`, because the name of the function must have been declared previously and be present in the environment. The formal parameters of the function however are passed to `NamerForDecls`, as these are variables implicitly declared, that have to be added to the current environment. This updated environment is then used by `NamerForBody` to transform the function's body. At the end, the parent's environment is restored, ensuring that formal parameters of the function are not available outside of the function.

This code also illustrates how functions can access variables available in the defining environment (such functions are called closures): all those are

inherited by the function's environment.

Some parts of the namer code are not that simple. For example `NamerForBody` declares a function `HandlePatternArgs` to apply the described transformation of functions and procedures with patterns arguments.

Handling `fCase` nodes also requires more work as it needs to

- declare new capture variables
- handle guards by, amongst other things, introducing a new symbol
- add guards corresponding to escaped variables in patterns

The function `NamerForCaptures` is called on the case patterns, collecting symbols to be declared in `Params.captures`. New symbols introduced for escaped variables are collected in `Params.guardsSymbols`.

As for the work of the namer in case of classes, it involves several steps as described in Section 3.1.2.5. Several functions are defined to help these steps. `NameMethodLabel` is the function used when traversing the methods list the first time to name only the method labels. `NameMethod` is used when traversing the methods the second time to name the arguments and the body.

Once these functions are defined, the namer's code for classes is quite simple, and listed in Appendix G to illustrate the different steps.

The code traversing `catch` patterns makes use of the same function used for naming `case` clauses, namely `NamerForCaptures`.

3.1.3 Desugar

The desugar pass handles the transformation of syntactic sugar code in its canonical form. Oz distinguishes two groups of instructions: statements and expressions. An expression is "syntactic sugar for a sequence of operations that return a value" [VRH04]. Statements do not return a value. The desugar function handles statements and expression differently, and defines two local functions, one for handling expressions, one for handling statements.

3.1.3.1 On statements and expressions

Handling expressions and statements is often very similar, but sometimes require significant differences of treatment. Hence the use two different local functions in the Desugar pass.

It is important that a lot of instructions can be used both as expression and as statement. Here is an example. In Oz, a cell is a mutable storage element which is created with an initial value (`C={NewCell 0}`), it can be given a new value (`C:=NewValue`) and its value can be read (`@C`). There's also an exchange operation `{Exchange C Old New}` which "atomically binds Old with the old content of the cell, and set New to be the new content" [VRH04].Assigning a new value to a cell is usually a statement returning no value, but when used as the right hand side of an unification, it becomes an expression, whose value is the Old value of the cell.

Determining if an instruction is a statement or and expression is done recursively from top to bottom, the starting case being that the program is a statement. In Figure 3.10, the whole code snippet which we consider in this case as a complete program is a statement. This `local...in...end` instruction

```
local
  C={NewCell 0}
  Old
in
  C:=1        % statement
  Old=(C:=2)  % expression, whose value is assigned to Old
  {Show Old}  % displays 1
end
```

Figure 3.10: Cell assignment as statement and as expression

is thus a statement. The declaration part of this instruction is always a list of `fSym`. The body is also a statement because the whole `local...in...end` is a statement. If it were an expression, the body would have been an expression. In our case, the body is a sequence of instructions, resulting in a hierarchy of `fAnd` tuples in the AST. Thus we see that a sequence can be a statement or an expression, in the latter case the value of the whole sequence is the value of the last instruction which must be an expression. This illustrates the need of distinct functions when statement and expressions must be handled differently.

If we note `F` the pass of the compiler, `FExpr` the locally defined function handling expressions, and `FStat` corresponding function for statements, we can ensure that the right function is applied to each instruction's AST as defined in section 2.3 by following these rules of identification, and call `FStat` on statements and `FExpr` on expressions:

program the program is a statement

fLocal
- the declarations are all just `fSym`s, since the `DeclsFlattener` has moved all other code to the body. These `fSym`s are neither expressions nor statements, they are just `fSym` declarations
- the body is of the same kind as its parent `fLocal`

fAnd
- the first feature is a statement
- the second feature is of the same kind as its parent `fAnd`

fEq both sides are expressions

fProc if the procedure's definition is an expression, its identifier must be the nesting marker $, else its identifier is an expression. The arguments are patterns, and the body is a statement except if one of its formal parameters has a nesting marker, in which case the body is an expression.

fFun are similar to `fProc`, because a function definition is desugared in a procedure definition as described in Section 3.1.3.3. The body of a `fFun` is an expression.

fApply calls can be statements or expressions, but in both cases the callee and its arguments are all expressions.

fColonEquals can be statement or expression, but `Cell` and `Val` are in both cases expressions.

fAt(Cell Pos) can only be used as an expression. `Cell` is an expression.

fRecord can only be used as expressions. Its label, features and values are expressions.

fCase can be a statement or an expression. The value tested is an expression. The `fCaseClause`s and the else code have the same type as their parent `fCase`. The patterns in the `fCaseClause`s are patterns, and the guards are expressions.

All expressions and statements in the AST will be visited by Desugar. Even if the statement itself is not transformed, as is the case for a unification, its children will be visited and possibly transformed. For example `X=5+2` will still be a unification after it's been desugared, but its children will have been transformed in `X={Number.'+' 5 2}`.

3.1.3.2 Operators

Operators are transformed in the call of their respective function as listed in Table 3.1.

Operator	Desugar result
+	Number.'+'
*	Number.'*'
-	Number.'-'
~	Number.'~'
div	Int.'div'
mod	Int.'mod'
/	Float.'/'
==	Value.'=='
>=	Value.'>='
=<	Value.'=<'
>	Value.'>'
<	Value.'<'
\\=	Value.'\\='
:=	Value.catExchange

Table 3.1: Operators and their desugar result

The operators are parsed as `fOpApply` tuples as described in section 2.3.9. Building further on the example of that section, the AST coming from the parser

```
fOpApply(
   '+'
   [ fInt(5 pos)
     fInt(2 pos) ]
   pos)
```

will be transformed in:

```
fApply(
   Number.'+'
   [ fInt(5 pos)
     fInt(2 pos) ]
   pos
)
```

3.1.3.3 Functions

Functions are transformed in their canonical procedure form. This is done simply by replacing the function by a procedure with exactly the same characteristics, except that it takes one additional argument (the return value), and the body of this procedure is the unification of this new argument with the original body of the function. This resulting AST is itself recursively desugared.

For example, the code

```
fun {F A B}
   A+B
end
```

will have its AST transformed such that it corresponds to this Oz code:

```
proc {F A B ?R}
   R={Number.'+' A B}
end
```

A new symbol, denoted `R` for convenience, is added to the arguments list of the procedure, and this new symbol is then unified with the body of the original function. This unification will in turn be handled by the unnester, which is the next pass of the compiler.

The function definition is parsed in the AST of Figure 3.11 and transformed by the desugar pass into the AST in Figure 3.12.

Lazy functions have in their flags a lazy atom: `fAtom(lazy pos)`. The desugar step of lazy function consists in wrapping in a thread a call to the builtin `waitNeeded` on the return symbol before unifying it with the body of the function. If the function of the previous example had been marked as lazy, the transformation would have yielded this result, in Oz notation:

```
proc {F A B ?R}
   thread
      {WaitNeeded R}
      R={Number.'+' A B}
   end
end
```

3.1.3.4 Procedures

The only special operation in desugaring procedure definitions, is the case procedures having the nesting marker as one of their formal parameters. At the call, the argument passed in place of the nesting marker will be unified with the body of the procedure which much be an expression. Understanding this makes the transformation immediate. For example, this procedure

```
fFun(
   fSym(F pos)
   [  fSym(A pos)
      fSym(B pos) ]
   fOpApply(
      '+'
      [ fSym(A pos)
        fSym(B pos)]
      pos)
   pos)
```

Figure 3.11: Function definition

```
fProc(
   fSym(F)
   [  fSym(A pos)
      fSym(B pos)
      fSym(R pos) ]
   fEq(
      fSym(R)
      fApply(
         fConst('+' pos)
         [  fSym(A pos)
            fSym(B pos)]
         pos)
      pos)
   pos)
```

Figure 3.12: Desugared function definition

```
proc {Succ In $}
   In+1
end
```

will be desugared in

```
proc {Succ In R}
   R=In+1
end
```

3.1.3.5 Calls

A node `fApply(Op Args Pos)` has `Op` and `Args` desugared as expressions.

3.1.3.6 Records

As mentioned in the description of records in Section 2.3.12, the features can be left out, and are in that case implicit, with increasing integer values starting from 1. The Desugar pass will make these features explicit.

In Oz syntax, it means that `rec(a b c)` is transformed in `rec(1:a 2:b 3:c)`. Of course, attention has to be paid by the programmer to avoid conflict, as implicit feature assignment does not check for explicit features'values, and conflict can arise. For example `rec(a 1:b c)` will be transformed in `rec(1:a 1:b 2:c)`. This behaviour is consistent with Mozart 1.

Records undergo two additional transformations. First, `fRecord` that have all label, features and values constant are replaced by a `fConst` with as value the record reconstructed. This greatly simplifies the AST for these records as is illustrated in the next example, and can be done because the value of the record can be used as is in later steps, notably the opcodes generation. This transformation is called constant folding.

```
fRecord(
   fAtom(rec pos)
   [
      fColon(
         fInt(1 pos)
         fAtom(a pos)
      )
      fColon(
         fInt(2 pos)
         fAtom(b pos)
      )
   ]
)
```

is transformed in

```
fConst( rec(1:a 2:b) pos)
```

Records with non constant labels and/or features are replaced by a call to `Boot_Record.makeDynamic`, `Boot_Record` being a builtin module (a module implemented in C++ and made available to the Oz code as if it was a functor). The goal of this transformation is to leave in the AST only records with constant label and features. `MakeDynamic` takes two arguments. First the label of the record to construct. And second, a tuple with label '#' and whose values are alternately the features and their respective value. As a result, we replace `Rec(F1:V1 F2:V2)` by

```
{Boot_Record.makeDynamic Rec '#'(1:F1 2:V1 3:F2 4:V2)}
```

We end up with an AST that only contains records with constant label and features: only the values are not constants. This is required because there is no opcode capable of handling a record with non-constant arity.

3.1.3.7 Pattern Matching

Records and open records present in a pattern undergo the same desugar transformation as normal records. Patterns however can only have constant label and features.

3.1.3.8 Wildcards

Although the programmer is not interested in these values, the compiler will declare and place a new synthetic symbol in each of these locations. In oz code

notation, `_` is replaced by the expression `local NewSymbol in NewSymbol end`. Patterns with guards, found in the AST as

```
fNamedSideCondition(Pattern Decls Guards GuardSymbol Pos)
```

have their features handled according their statement/expression character: `Pattern` is a pattern, `Decls` are left as is, `Guards` is a statement (see the introduction of `fNamedSideCondition` by the namer in Section 3.1.2.3) and `GuardSymbol` is just a symbol, left as is.

Pattern conjunctions, present in `fPatMatConjunction` records, have their features desugared and are then placed in a secure structure like `fSym` in patterns are by the namer (Section 3.1.2.3). This is done because no other transformation should be done on the pattern conjunction's member before it is used by the code generator.

3.1.3.9 Threads

A thread expression `thread Body end` is desugared in the resulting statement

```
local
   X
in
   thread X=Body end
   X
end
```

This statement is then itself desugared.

Transforming a thread statement consists in the creation of a procedure with no argument whose body is the desugared body (which is a statement) of the thread, and then passing this procedure to the internal `Thread.create` procedure. In Oz notation, `thread Body end` is transformed in

```
local
   P
in
   proc {P}
      DesugaredBody
   end
   {Thread.create P}
end
```

At the AST level `fThread(Body Pos)` is transformed in this AST (taken from the compiler source code):

```
fLocal(NewProcSym
       fAnd( fProc(NewProcSym nil {DesugarStat Body Params} nil pos)
             fApply(fConst(Boot_Thread.create pos) [NewProcSym] pos)
           )
       Pos)
```

3.1.3.10 Locks

Lock instructions are handled very similarly to thread statements: their body is wrapped in a argumentless procedure, which is passed as argument, together with the lock itself, to the base environment's `LockIn`. In Oz notation, `lock L then Body end` is transformed in

```
local
   P
in
   proc{P}
      Body
   end
   {LockIn L P}
end
```

3.1.3.11 If then else

The condition tested is always an expression. The instruction itself can be a statement as in

```
if Cond then {DoTrue} else {DoFalse} end
```

in which case both code branches are desugared as statement or an expression as in

```
X=if Y>Z then bigger else smallerOrEqual end
```

in which case both code branches are desugared as expression. A `fNoElse` tuple is left as is and will only be useful for the opcode generation function.

3.1.3.12 Case

The case instruction is very similar to `if..then..else..end`. The condition tested is always an expression. The `case` instruction can be a statement as in Figure 3.13 , in which case clauses are desugared as statement or an expression as in Figure 3.14 in which case each clause is desugared as an expression.

```
case Rec
of action(A) then
   {DoAction A}
[] 'skip'() then
   skip
[] else
   {LogError}
end
```

Figure 3.13: Case statement

```
PersonId= case Rec
          of create(Name) then
             Person = {Create Name}
          in
             Person.id
          [] find(Name) then
             Person = {Find Name}
          in
             if Person==unit then
                unit
             else
                Person.id
             end
          [] else
             unit
          end
```

Figure 3.14: Case expression

As for the `if..then..else` instruction, the absence of an else clause is indicated by a tuple `fNoElse` in the AST. If no else clause was present, the desugar step introduces one which raises an error, transforming

```
case Rec
of action(A) then
   {DoAction A}
end
```

in

```
case Rec
of action(A) then
   {DoAction A}
else
   {Boot_Exception.'raiseError' kernel(noElse File Line Rec)}
end
```

3.1.3.13 Booleans Combinations

Conjunctions of boolean expressions with `andthen` are desugared in `if..then..else` expressions. Here is the translation rule: `Cond1 andthen Cond2` is translated in `if Cond1 then Cond2 else false end` We see that conditions are evaluated from left to right, and that conditions evaluation stops as soon as possible, i.e. when a false condition is encountered.

`orelse` combinations are similarly translated from `Cond1 orelse Cond2` to `if Cond1 then true else Cond2 end` The example of Figure 3.15 is transformed in the code in Figure 3.16.

becomes

This form will then be handled by later passes of the compiler.

```
(A==1 andthen B==2) orelse C==3
```

Figure 3.15: `orelse` code to be desugared

```
if (if A==1 then B==2 else false end)
then
  true
else
  C==3
end
```

Figure 3.16: Desugaring `orelse`

3.1.3.14 Classes

Handling classes in the compiler consists in desugaring the class definition in a call to `OoExtensions.'class'`. The AST node `fClass(FSym Specs Methods Pos)` gets transformed in

```
fApply(fConst(OoExtensions.'class' Pos)
       [Parents Meths Attrs Feats Props PrintName FSym]
       Pos)
```

The `Desugar` function just has to correctly build the arguments of the call. Of course, the arguments are themselves ASTs. So an argument that is a list will be in fact an `fRecord`.

Information to build the elements `Parents`, `Attrs`, `Feats` and `Props` come from the `Specs` value in the `fClass` node.

`Parents` is simply a list of the parents of the class. The elements of the list are simply the `fSym` tuple holding the symbols corresponding identifying the parent classes.

`Meths` is a #-tuple with one item of the form `name#Proc` for each method, where `name` is the method name, and Proc is a procedure (which we will characterise as a "proxy" procedure) with 2 arguments implementing the method. The two arguments of the method are `self`, which is the object on which the method is called, and `M`, the message corresponding to the call. It is the message that contains all argument values. It is important to note that the symbols used inside the body of the method don't correspond to the arguments of the procedure implementing the method. This mismatch is handled by unifying symbols in the method body with values found in the message. Figure 3.18 shows a methods and its implementing procedure.

Of course, the compiler handles keyed arguments, where the features of the message `M` are not only numeric.

As described in Section 2.3.19 on page 22, the message corresponding to the method call can be captured and made available to the body of the method. The message being the second argument of the "proxy" procedure, it is easy to make it available in the method body by prepending it with a unification of the header capture symbol and the message symbol itself. A better approach might be to use the capture symbol directly as the message symbol in a future revision of the code.

```
meth foo(X Y)
   {Show X+Y}
end
```

Figure 3.17: Example method

```
proc {Foo Self M}
   X Y
in
   X = M.1
   Y = M.2
   {Show X+Y}
end
```

Figure 3.18: Procedure implementing example method

Handling default values needs special care. The unification of the symbol with its default value may only be done if no value was provided in the message `M`. But this can only be done at run time, hence the compiler has to inject code in the AST to check if a value was provided for the method argument. Here is the code injected in the AST, where `Index` is the index of the argument that is handled and `MessageSymbol` is the symbol corresponding to the message of the method call:

```
fEq(Sym
    fBoolCase(fApply(fConst(Value.hasFeature pos)
                     [MessageSymbol fConst(Index pos)] pos)
              fOpApply('.' [MessageSymbol fConst(Index pos) ] pos)
              Default
              pos)
    pos)
```

which, written in Oz, is:

```
Sym=if {Value.hasFeature MessageSymbol Index} then
       MessageSymbol.Index
    else
       Default
    end
```

Note that this code behaviour is different from

```
{CondSelect MessageSymbol Index Default}
```

as in the latter case, all arguments of `CondSelect` will be evaluated first.

The body of the method is desugared with an additional bit of information compared to normal procedures: the symbol corresponding to `self`, the object on which the method is called. The availability of `self` is not the only difference in the desugarisation of the methods' bodies. Some operators also have another meaning inside methods, like for example `@`. In code outside methods, `@` is desugared in a call to `Boot_Value.catAccess`, but in methods, it has to be desugared in a call to `Boot_Value.catAccessOO`. There is a similar change for other operators like `:=` and the exchange operation. Some operators are

only available in methods, such as the attribute assignation operator `<-`. It is desugared in `{BootObject.attrAssign Self LHS RHS}`.

Just as procedures, methods can have the nesting marker as formal parameter. The transformation needed is exactly the same as explained in Section 3.1.3.4 for functional procedures.

`Attrs` is a record with label `attr`. For each attribute of the class, there is a feature-value pair in that record. The feature is the name of the attribute, the value is the default value provided in the class definition, or the result to a call to `{Boot_Name.newUnique 'ooFreeFlag'}`

`Feats` is a record with label `feat` built exactly in the same way as `Attrs`, but with the features present in the class definition.

`Props` is similar to `Parents` in that it is simply a list of properties of the class (`final`, `locking`). The `locking` property implicitly declares a lock object for the class, which can then be used with the `lock` instruction, which then appears in the AST as `fLock(Body Pos)`, which differs from the nodes `fLockThen` described in Section 2.3.15, notably because it doesn't take the lock object as argument. `fLock` needs to be desugared to make the lock object explicit. Instance an instance of a class, the lock object can be obtained by a call to `OoExtensions.'getObjLock'`, which can then be used in a `fThenLock` instruction. In practice, the node

```
fLock(Body Pos)
```

will be desugared in

```
fLockThen( fApply(fConst(OoExtensions.'getObjLock' Pos)
                  [@(Params.'self')]
                  Pos)
           Body
           Pos)
```

which is itself desugared.

`PrintName` is an atom which is the name of the class, or `''` if it is an anonymous class.

This is all the compiler has to do to support classes. From this point, the AST has no more references to classes. The virtual machine will handle everything related to classes for us.

3.1.3.15 Exceptions

`fRaise` nodes are desugared in a call to the builtin `Exception.'raise'`. The exception is an expression, and desugared as such, transforming

```
fRaise(E pos)
```

in

```
fApply(
       fConst(Exception.'raise' pos)
       [ {DesugarExpr E Params} ]
       pos)
```

As a reminder, here is the AST of a try instruction:

```
fTry(
   Body
   fCatch(
      |(
         fCaseClause(
            Pat1
            Code1)
         |(
            fCaseClause(
               Pat2
               Code2)
            nil))
      pos)
   FinalCode
   pos)
```

Desugaring an `fTry` node without `finally` code is not much more complex. If the try instruction is an expression (resp. statement), the body and the clauses' code are desugared as expressions (resp. statements). A `case` instruction is introduced, using the very same clauses as the catch, transforming [BaseLang]

```
  try SE1
  catch C1 [] ... [] Cn
  [[ finally S ]]
end
```

in

```
try SE1
catch X then
   case X of C1 [] ... [] Cn
   else
      raise X end
   end
[[ finally S ]]
end
```

As this code shows, the very same clauses can be reused for the case instruction, without the need of complex AST transformation.

Note that an `else` clause is introduced in the `case` instruction. This is simply to re-raise the exception if no pattern matched, meaning that the exception was not caught at this level. In the transformed code, note also that a new symbol, `X` is introduced. This results in the new AST transformation for a try expression

from

```
fTry(Body fCatch(Clauses CatchPos) fNoFinally Pos)
```

to

```
fLocal(  NewSymbol
         fTry({DesugarExpr Body Params}
              fCatch([fCaseClause(NewSymbol
                                  {DesugarExpr fCase(NewSymbol
                                                     Clauses
                                                     fRaise(NewSymbol Pos)
                                                     CatchPos)
                                               Params})]
                     CatchPos)
              fNoFinally
              Pos)
         Pos)
```

Desugaring an `fTry` node with a `finally` block is more involved, and although the logic is similar, handling try-finally expressions and statements requires some differences of treatment.

The transformation arises from the fact that the `finally` code has to be executed, even when an exception was raised. Let's examine the transformation of `try` statements, which transforms this

```
try Body
finally S
end
```

to

```
local X in
   X = try
          try Body end
          unit
       catch Y then ex(Y)
       end
   S
   case X of ex(Z) then
      raise Z end
   else skip
   end
end
```

This example does not include `catch` clauses for brevity. They can be considered as being part of `Body`. A new variable (`X`) is introduced and unified with a new try expression whose body is a sequence made of the original `try` statement without its `finally` code followed by `unit`. This means that if execution of the original `try` statement does not raise an exception, the variable `X` will have the value `unit`. The new try catches all exceptions (which can only be exceptions not caught by the original try expression) with one clause whose body is a record with label `ex` holding the caught exception as value.

This construction ensures that the `finally` code can be executed, even if there is an exception raised not caught by the original try.

After the `finally` code is executed, it is needed to check if the original `try` executed successfully, in which case the `X` variable has value `unit`, or if it failed,

in which case the variable `X` is bound to a record with label `ex` holding the exception. In the latter case, the exception needs to be re-raised held in the record needs to be re-raised.

If the statement has no `catch`, but a `finally` block, the inner `try` is replaced by the body of the original `try` statement, resulting in a very similar structure:

```
  local X in
     X = try
            Body
            unit
         catch Y then ex(Y)
         end
     S
     case X of ex(Z) then
        raise Z end
     else skip
     end
end
```

We have seen that the original `try` statement is transformed in an expression by putting it in a sequence which second element is simply the `unit` expression. One might propose to handle `try` expressions in the same way, and test if the value bound to `X` is a record with label `ex` to know if an exception was raised. But we have to accept a `try` expression whose value is a record with label `ex`. The consequence is that the transformation applied in case of `try` expressions is very similar, but with a different value for the new `try` expression. The body of the new `try` expression is simply a record of label `ok` and holding as value the original try expression. It will thus transform this expression

```
  try Body
  finally S
  end
```

to

```
  local X in
     X = try
            ok(try Body end)
         catch Y then ex(Y)
         end
     S
     case X of ex(Z) then
        raise Z end
     else X.1
     end
end
```

Wrapping the value of the original `try` expression in a record with a specific label (`ok` in our case) enables us to check, after execution of the `finally` code, if the execution was successful or not. If an exception was raised, it is re-raised. If no exception was raised, the expression's value is extracted from the success record and returned.

3.1.3.16 Array assignation

The node `fDotAssign` found in the AST is of the form

```
fDotAssign(fOpApply('.' [LHS CHS] Pos1)
           RHS
           Pos2)
```

and is desugared in

```
fApply(fConst(Boot_Value.'dotAssign' Pos1) [LHS CHS RHS] Pos2)
```

which is itself recursively desugared. See section 3.1.3.5 to see how an `fapply` node is desugared.

3.1.3.17 Functors

Anonymous functors are expressions, and named functors are statements but the transformations described below are applied in exactly the same way on expression and statement functors.

A different treatment is applied to compiled and computed functors, which differ as explained in Section 2.3.23.

A compiled functor is desugared in a call

```
{NewFunctor [ImportRecord ExportRecord ApplyFun]}
```

taking three arguments:

ImportRecord is of the form illustrated in Figure 3.19 where `ModName` is the name of the variable identifying this imported module, and `AliasesAtoms` is the list of atoms identifying the imported functions. An example is given in Figure 3.20.

ExportRecord is a record with label `'export'`. Its features are the exported keys, and the values of each feature is the atom `value`, stating that the values can be of any type. This information is currently not used further by the compiler. An example of an ExportRecord is given in Figure 3.21.

ApplyFun is a function taking one parameter, as shown in Figure 3.22. When a functor is applied, this function is called and it is the record it returns that gives entry points to the module functionality. The structure of this function can be put in relation with the functor variables scopes illustrated in Figure 3.8 on page 39.

```
import(ModName:info(type:AliasesAtoms
                    from:"x-oz://system/"#ModName#".ozf"))
```

Figure 3.19: `ImportRecord` argument of `NewFunctor` call

Computed functors are desugared differently. A computed vector has at least one `require` or `prepare`. It is desugared in a new outer functor itself defining an inner functor. This is best illustrated by the example of a functor in Figure 3.23 and its desugared result in Figure 3.24.

As the outer functor is immediately applied, its `import` and `define` sections are evaluated. But these are, or include, the original functor's `require` and `prepare` sections, and this construction ensures the behaviour we need. This result is recursively desugared.

This import directive:
```
import
   DumpAST(dumpAST PPAST) at '../lib/DumpAST.ozf'
```
will result in this import record definition:
```
import('DumpAST':info(type:[dumpAST] from:'../lib/DumpAST.ozf'))
```

Figure 3.20: `ImportRecord` example

This export directive:
```
export
   dumpAST:DumpAST
```
will result in this import record definition:
```
export(dumpAST:value)
```

Figure 3.21: `export` directive example

```
fun {$ ImportParamSym}
   local
      <importedSymbolsDeclarations>
      <defineDeclarations>
   in
      <importBinds>
      <defineStats>
      'export'( dumpAST:DumpAST ...)
   end

end
```

Figure 3.22: `ApplyFun` structure

3.1.3.18 Compiler Code Notes

The desugar pass is implemented by a function named `Desugar`. It defines multiple local functions:

DesugarOp called to desugar operators

DesugarRecordFeatures working on record features, making them explicit if need be, and also desugaring the value corresponding to each feature.

TransformRecord called to transform records as described above, just after it has been desugared.

IsConstantRecord called by `TransformRecord` to determine if the record for which the AST is passed as argument is a record with constant label, features and values

HandleLazyFlag will wrap the body of a function in a thread if needed, as described in Section 3.1.3.3.

DesugarExpr called to desugar expressions

```
functor
require <req>
prepare <prep>
import <imp>
export <exp>
define <def>
end
```

Figure 3.23: Computed functor example

```
local
   functor Outer
   import <req>
   export inner:Inner
   define
      <prepareDecls>
   in
      functor Inner
      import <imp>
      export <exp>
      define
         <defDecls>
      in
         <defStats>
      end
      <prepareStats>
   end
in
   {ApplyFunctor BaseURL Outer}.inner
end
```

Figure 3.24: Desugared computed functor

DesugarStat called to desugar statements

DesugarClass is handling class definitions

The program itself being a statement, the function Desugar simply call DesugarStat on the AST of the whole program. This will trigger further calls to DesugarExpr and DesugarStat that will traverse and transform the whole AST.

`DesugarClass` defines multiple local functions:

TransformMethod Takes one method signature and an index, and builds its element of the '#' record containing the class' methods. The Body is also desugared with the SelfSymbol passed in Params. This is needed, for example for desugaring `@attribute` in `catAccessOO` rather than in `catAccess`. In this case, `DesugarOp` will check if `self` is available to decide which result to produce. Functional methods, defined with one of their formal parameters being the nesting marker, are also handled by this function, using functions `HandleDollarArg` and `InjectDollarIfNeeded` also used to

work on function procedures. As we've seen, several constructs (like parameters default values, method head capture, nesting marker as formal parameter) need to inject symbol declarations and initialisations in a method's body. It is `TransformMethod` that manages all this, constructing a list of pair `Symbol#InitCode` which is then used to inject declaration and initialisation code in the body of the method.

TransformAttribute sets default value `{Boot_Name.newUnique 'ooFreeFlag'}` if none was provided.

With these functions defined, the code of `DesugarClass` is greatly simplified and easy to understand.

3.1.4 Unnester

The unnester will result in an AST in which there are only elementary instructions. For example, after the unnester, the arguments of functions all are symbols. The instructions susceptible to be unnested are inspected each in turn in the following subsections.

3.1.4.1 Unification

The unnesting result of a unification instruction depends on the elementary character of each side. `fConst` and `fSym` are considered elementary AST nodes, as these are not to be unnested in a unification. Considering the elementary character of both sides of the unification, we have to handle three situations:

Both elementary In this case there is nothing to do

One complex, one elementary The unnesting is done according to what the complex expression is:

Call This is either a call without nesting marker in a pattern position, `V={P E1...En}`, which needs to be transformed in the form `{P E1...En V}`, i.e. the elementary side is injected as last argument of the call. Or it is a call with the nesting marker in a pattern position, of the form `V={P E1..$..En}`, which needs to be transformed in `{P E1..V..En}`.

Sequence of instructions In the AST, the sequence of instructions is available in a hierarchy of `fAnd` tuples. We also know that the value of a sequence of instructions is the value of the last instruction in the sequence. As a consequence, the unnesting of this type of unification is done by moving the unification to the second feature of the fAnd, and recursively unnest the resulting AST.

Declaration (local) The value of a `local..in..end` in a unification is the value of its body. Hence, the unification

```
V = local declarations in body end
```

is moved inside the `local`'s body resulting in the code

```
local declarations in V=body end
```

This clearly requires a recursive call as the body can be a sequence of instructions, which will have to be handled as described previously.

if then else The unification is simply moved inside each branch, the symbol being unified with the branch body and the resulting code is further unnested. For example, `V=if B then Is1 else Is2 end` is transformed in `if B then V=Is1 else V=Is2 end`. This needs to be recursively unnested as Is1 and Is2 can be, for example, sequences of instructions.

case Similar to the previous item, the elementary side unified with the case expression is unified in each clause with its respective body.

try The unification is moved inside every `catch` clause, but not in the `finally` code. It needs extra precaution though and is analysed specifically in Section 3.1.4.2.

Anonymous procedure The elementary side replaces the fDollar, for example `P = proc {$ A} Body end` is transformed by replacing `$` by `P` yielding `proc {P A} Body end`

Both complex This is unnested by creating a new synthetic symbol, unifying it with the left side then with the right side, and unnesting the resulting code.

In the case of `fApply` the unified symbol may have to be injected in place of a nesting marker deeply nested in a record argument, like in this example:

```
{GetAttribute Val attr(value:$ ...)}
```

where we suppose that `GetAttribute` has the signature

```
proc {GetAttribute Object Rec}
```

To handle this case, arguments have to be tested to see if a nesting marker is present, and the code needs to descend in the record arguments. Once the unified symbol is injected in the arguments list, a recursive call is done to unnest the `fApply` itself.

This part of the code requires enhancements, as it does not compile this code successfully due to the non-constant feature:

```
local
   A P
in
   A=a
   proc {P ?R}
      R=rec(a:1 b:2)
   end
   {Show {P rec(A:$ b:2)}}
end
```

All cases covered by the unnesting of unifications make recursive calls on their result. Here is an illustration of the need of recursive calls. It also explains why this particular form executes successfully:

```
3+4 = {Show}
```

It is transformed due to both sides being non-elementary, in this form in a first step:

```
local
   T
in
   T = 3+4
   T = {Show}
end
```

The unnester will work on the already transformed code

```
local
   T
in
   T = {Number.'+' 3 4}
   T = {Show}
end
```

and finally in this form by the transformation of the assignment of a procedure call in the same procedure call with the return variable passed as additional argument:

```
local
   T
in
   {Number.'+' 3 4 T}
   {Show T}
end
```

The order of the assignments of the first step explains why this doesn't work:

```
{Show} = 3+4
```

3.1.4.2 Exceptions

`raise` expressions at this stage are present in the form `Sym=raise E end`, and this unification statement is simply replaced by `raise E end`

A `try` expression at this stage is found in the form

```
Sym = try Body
    [ catch y then CatchExpr ]
    end
```

and the unnesting simply consists of injecting the unification inside the `try`'s body and inside each clause, resulting in:

```
try X in
   X = Body
   Sym = X
[ catch y then Sym = CatchExpr ]
end
```

There is a subtlety though, in that the symbol originally unified with the `try` expression, when injected in the `try`, is not directly unified with the original body. This is to account for the special case of a procedure raising an exception after having bound `Sym`. Here is an illustration, actually from the test suite of the compiler (see test 372):

```
local
   proc {P ?X}
      X = 42
      raise exception end
   end

   Y =
   try {P}
   catch exception then
      error
   end
in
   {Show Y}
end
```

With a single assignation, the transformation would result in

```
local
   proc {P ?X}
      X = 42
      raise exception end
   end

   try {P Y}
   catch exception then
      Y=error
   end
in
   {Show Y}
end
```

In this case `Y` would be bound by the procedure to 42, before it raises an exception. But when the exception is caught, `Y` is again assigned, this time with `error`.

The double assignation solves this problem, by producing this code:

```
local
   proc {P ?X}
      X = 42
      raise exception end
   end

   try
      X
   in
     X={P}
     Y=X
   catch exception then
      Y=error
   end
in
   {Show Y}
end
```

When `P` raises its exception, it hasn't bound `Y` but the proxy variable `X`. Once the statement `Y=X` is reached, no exception can be raise anymore, and

there is no risk of executing the assignation `Y=error` in the `catch` clause.

Remember that the statement produced is then recursively unnested.

3.1.4.3 Calls

All arguments are examined in turn, one by one. Elementary arguments are left untouched Complex arguments are extracted from the arguments list by:

1. declaring a new symbol.
2. unifying this new symbol with the argument
3. replacing the argument by the new symbol in the argument list.

Here is an example of unnesting in Oz code:

```
{F {F2 A} B}
```

will be transformed in:

```
local
  NewVar
in
  NewVar={F2 A}
  {F NewVar B}
end
```

If multiple arguments are complex, all are declared in the same `local..in..end` instruction.

Of course, the function called can itself be the result of a function call, and this is handled too:

```
{ {F1 A} B }
```

will result in an AST for this code:

```
local
  NewVar
in
  NewVar = {F1 A}
  { NewVar B }
end
```

If a nesting marker is found in a record argument, it is replaced by the other side of the unification, transforming

```
Res={P rec($)}
```

into

```
{P rec(Res)}
```

3.1.4.4 Declarations (local)

The unnester will not touch at the declaration part of a `local..in..end` instruction. It will only traverse the body.

3.1.4.5 If then else

The unnesting of the instruction `if Cond then Branch1 else Branch2 end` depends on the type of Cond.

- If `Cond` is elementary, the unnesting is done simply by unnesting the two branches.
- If `Cond` is complex, a new symbol is created and unified with it. This is then followed by the `if..then..else` instruction in which the complex condition is replace by the symbol newly introduced:

  ```
  if Cond then Is1 else Is2 end
  ```

  becomes

  ```
  local
     S
  in
     S=Cond
     if S then Is1 else Is2 i
  end
  ```

Looking at the result of the Desugar pass for the boolean combination example on Figure 3.16 page 50, this code

```
if (if A==1 then B==2 else false) then
  true
else
  C==3
end
```

Will become

```
local
   Cond
in
   Cond=(if A==1 then B==2 else false end)
   if Cond then
     true
   else
     C==3
   end
end
```

and then, by unnesting the `if..then..else`:

```
local
   Cond
in
   if A==1 then Cond=(B==2) else Cond=false end
   if Cond then
     true
   else
     C==3
   end
end
```

There is a notation abuse in these last examples, as `Cond=(B==2)` should have been written `{Value.'==' B 2 Cond}` to be correct. This was not done to focus on the transformation at hand.

3.1.4.6 Case

Unnesting `case` instructions is very similar to unnesting `if..then..else` instructions. If the value tested is not elementary, it has to be extracted and replaced by a new symbol unified with it.

```
case {Label R}
of l then
   {Show 'L record'}
end


local
   V
in
   V = {Label R}
   case V
   of l then
      {Show 'L record'}
   end
end
```

3.1.4.7 Records

The values in a record can be of non-elementary form, for example a function call. This has to be extracted from the record definition and replaced by a new symbol which has been unified with the initial value.

The work happening on the records is very similar to the work done on (function and procedure) calls. Rather than working on the procedure arguments, we work on the pairs feature-value and only handle the value.

There is a catch however: records in tail positions of a procedure need to be unnested differently. To understand why, let's first look at how records are unnested when they are not in tail position.

Here is an example where the value of a feature is an if expression.

```
R=rec(a:if B then 2 else V end b:2)
```

Supposing the record is not in tail position, it will be transformed in two steps. First non-elementary value will be extracted from the record, resulting in

```
local
  NewVar
in
  NewVar=if B then 2 else V end
  R=rec(a:NewVar b:2)
end
```

And this requires a new unnesting step because the right side of the unification is an `if` expression (see the unnesting of unifications Section 3.1.4.1). This

second transformation results in

```
local
  NewVar
in
  if B then NewVar=2 else NewVar=V end
  R=rec(a:NewVar b:2)
end
```

Figure 3.25 lists code with a record in tail position.

```
fun {Fill V}
   {Delay 5000} % Represents long computation
   if V>1000 then
           nil
   else
      V+1|{Fill V+1}
   end
end
```

Figure 3.25: Producer with record in tail position

The expected behaviour is that the consumer of the items can start to work as soon as there is one item inserted. Such a consumer can be the procedure of Figure 3.26.

```
proc {Print L}
   case L
   of X|Xs then
      {Show X}
   [] nil then
      skip
   end
end
```

Figure 3.26: Consumer procedure

However, if the record `V+1|{Fill V+1}` is desugared in

```
V1=V+1
V2={Fill V+1}
Ret=V1|V2
```

The record will be accessible only after the statement `V2={Fill V+1}`. This means that the consumer will only be able to start working only when the producer returns. The solution is to unnest it like this:

```
Ret=V1|V2
V1=V+1
V2={Fill V+1}
```

In that case, the consumer will be able to start working as soon as `V1` is bound, i.e. as soon as `V1=V+1` is executed.

Another illustration is given by the Oz code in Figure 3.27. Figure 3.28 shows the situation if the unnesting is done incorrectly: the return value will be available only after `V2` is available, i.e. only after `MyMap` has traversed the whole list.

```
fun {MyMap L F}
   case L
   of X|Xs then
      {F X}|{MyMap Xs F}
   [] nil then
      nil
   end
end
```

Figure 3.27: Record in tail position of MyMap function

```
   V1={F X}
   V2={MyMap Xs F}
   Ret=V1|V2
```

Figure 3.28: Wrong unnesting in tail position in MyMap

Determining if a record is in tail position is done top-down. The body of a procedure is in tail position. For `fAnd(First Second)`, `First` is not in tail position, and `Second` is in tail position only if its parent `fAnd` was in tail position. Same reasoning applies for clauses of a `fCase` and branches of a `fBoolCase`.

3.1.4.8 Nesting Marker

It is also the unnester that will handle the nesting marker and remove it from the AST. There are three cases that are handled and have been covered. Here is a summary:

Definitions of procedures At this step, if we found a nesting marker as the identifier of a procedure, it is necessarily in an assignment statement. The unnester needs to transform `X=proc{$ A..N}` in `proc{X A..N}`.

As argument of a call In this case, the nesting marker has to be replaced by the variable the expression is unified with. For example, `X={P A $ B}` needs to be transformed in `{P A X B}`.

Inside a pattern argument This can be considered as a more general case of the second bullet. The nesting marker is replaced by the symbol the expression is unified with.

3.1.4.9 Compiler Code Notes

The unnester is implemented by a function called `Unnester`. The unnester implements multiple local functions:

- `IsElementary` will return `true` if the AST passed as argument is elementary.
- `UnnestFEq` is called to unnest unification ASTs
- `BindVarToExpr` is the function called by `UnnestFEq` when one side is elementary, and the other complex.
- `UnnestFApply` is called to unnest calls
- `UnnestFRecord` is called to unnest records and open records. Its structure is similar to that of `UnnestFApply`.
- `UnnestFBoolCase` is called to unnest `if..then..else` instructions.

If the unnester encounters a tuple in the AST for which no specific operation is prescribed, it recursively unnests each feature of the record. An improvement to the code would be to explicitly handle every kind of tuple found in the AST.

The unnester also tracks if an instruction is in tail position or not, simply by maintaining a `tail` field in the `params` record.

3.1.5 Globaliser

This pass handles the global variables of a procedure, that is, variables used by a procedure that it does not declare (implying that the declaration is done at an upper level in the AST). It is interesting to note that the concept of global variables is only defined for procedures. Just as locals attached an environment to their body, procedures attach their own unique procId to their arguments, declarations and body. Symbols for variables declared by a procedure, including formal parameters, get assigned the procId of said procedure.

Here is a simple example of a procedure having one global variable, `A`:

```
A=1
proc {P}                % A is global to P
   B=2
in
   {Show A+B}
end
```

When `P` is executed, it will show 3, using its locally declared variable `B`, and its global variable `A`.

Special attention has to be paid to nested procedure declarations. The globals of procedures in the code have to be determined from the inside to the outside: determine the globals of the deepest nested procedure, and go up. The globals of a procedure are:

- all variables it uses directly (i.e. not in a procedure definition)
- plus the globals of all the procedures it defines
- minus the variables it declares itself

In the example of Figure 3.29 we see that `P3` uses 2 variables declared by another procedure: `A` and `B`. `P2` does not directly use any variable, but it defines `P3` which itself has 2 global variables. Those two variables are thus also globals

```
A=1
proc {P1}                  % A is global to P1
   B=2
   P2
in
   proc {P2}               % A and B are globals to P2
      P3
   in
      proc {P3}            % A and B are globals to P3
         C
      in
         {Show A}
         {Show B}
         {Show C}
      end
      {P3}
   end
   {P2}
end
```

Figure 3.29: Example of nested procedures

to `P2`. `P1` also does not use any variable directly, but it gets 2 globals from `P2`, of which it declares one: `B`. `P1` indeed only has one global: `A`.

In the AST, we replace symbols corresponding to a global variable by a new local symbol of type `localised` referencing the symbol for this variables in the parent proc which might itself be a `localised` symbol, and so on until we reach the level where the variables is declared. `fProc` is also replaced by `fDefineProc` taking 1 additional argument: the newly created local variables referencing a variable in the surrounding procedure. This additional information will enable the code generator to handle global variables.

Handling globals correctly requires to handle numerous cases, as illustrated by these examples:

```
local
   A
in
   proc {P1}
      P2
   in
      proc {P2}
         P3
      in
         proc {P3} {Show A} end
         {P3}
      end
   end
end
```

In this case, `A` is global to `P3` and it creates a new local symbol that will reference a symbol local to `P2`. But for `P2`, A is also global, and this situation needs to trigger a new local creation in `P2`, which will be referenced by the new local in

P3. Same thing in P1.

```
local
   A
in
   proc {P1}
      P2
   in
      {Show A}  % Use A before a defined procedure needs it
      proc {P2}
         P3
      in
         proc {P3}
            {Show A}
         end
         {P3}
      end
   end
end
```

In this case, the traversal of the AST will first analyse `{Show A}` and create a new local symbol for the variables `A` in `P1`. `P2` is visited later and will create a new local to reference a symbol local to `P2`, which is also a new local as `A` is global to `P1`. But this new local has already been created, and this situation must not trigger the creation of a new local symbol for `A`. Rather, the new local symbol to `P2` must reference the new local symbol to `P1`. We see that the new local variables created in a procedure definition may have to be modified by its parent procedure.

Of course, the inverse has to be handled too, i.e. a new local is created for a defined proc, and reused later:

```
local
   A
in
   proc {P1}
      P2
   in
      proc {P2}
         P3
      in
         proc {P3}
            {Show A}
         end
         {P3}
      end
      {Show A}  % Use A after a nested proc triggered the creation
                % of a new local.
   end
end
```

Also, a new local symbol must not be created when it would represent a locally declared variable, as in this case:

```
local
   A
in
```

```
   proc {P1}
      P2
   in
      proc {P2}
         P3
      in
         proc {P3}
            {Show A}
         end
         {P3}
      end
   end
   {Show A}  % A is defined locally, and the nested procs defined
             % should not trigger the creation of a new local, but P1's
             % new local for A should reference the A declared by
             % the local .. in .. end
end
```

Two procedures defined at the same level must have have their respective new local symbols reference the same symbol at the parent level (whether it be a new local symbol created at the parent level or the existing symbol of a declared variable at the parent level). In this example, `P2` and `P3` each create a new local for `A` and both trigger the creation of a new local in `P1` for `A`. But only one new local should be created in `P1` for `A`, and both locals in `P2` and `P3` should reference it.

```
local
   A
in
   proc {P1}
      P2 P3
   in
      proc {P2}
         {Show A}
      end
      proc {P3}
          {Show A}
      end
      {P2}
      {P3}
   end
   {Show A}  % Use A after a nested proc
             % triggered the creation of a new local.
end
```

Figure 3.30: Use of a local variable after a nested proc definition accesses it as global

In the end, the algorithm used is the following. Each call of the globaliser takes the AST sub tree and an additional `Params` argument with three fields:

- the current procId

- a list of global variables already seen in the current procId
- a list, each item being the list of new local symbols created for the global at the same position in the previous list.

The globaliser handles the following nodes:

fProc When handling a `fProc`, the globaliser call gets the informations from the parent in its `Params` argument. Because we enter a new level of procedure nesting, a `NewParams` is initialised to be passed to recursive globaliser calls handling children nodes. `NewParams` is initialised with a new procId and two empty lists.

The symbol of the procedure itself gets the procId of its parent, found in `Params`. The arguments of the procedure get the procId of the currently handled procedure, found in `NewParams`. At that time the globaliser function is called recursively on the children nodes, with `NewParams` as additional argument.

Once all children have been traversed, the list of their globals and their respective newly created locals is found in `NewParams`. We have multiple new locals corresponding to one global in the case of 2 sibling procedure definitions referring the same global as in Figure 3.30.

Globals whose procId match the `NewParams`' procId are ignored, as those are variables that are declared at this level, and the chain of references must stop here for these variables.

For each remaining global, we look at the list of their locals. If one of these has the procId matching the current level, all other locals are changed to reference this one. This is because this specific symbol has been created by a direct use at this level, and we will use this one as the new localised symbol at this level. Then we push the global on the parent's globals list and add the local symbol to its corresponding list of new locals (those lists are found in `Params`)

If none of the locals has the current level's procId, it means the global variables has only been used in nested procedure definitions. We need to create a new local symbol for that variable at the current level that will be referenced by the existing locals. Finally, we push the global and the new local symbol we created to the parent via `Params`.

fSym If the symbol in this `fSym` has the current procId found in `NewParams`, keep it as is, do not change it. If its procId is different from the current procId, this `fSym` represents a global variable for the current level. We look in `Params` if a local variable with the current procId has already been created for this global variable. If yes we reuse it, else we define a new symbol, having the current procId and referencing the symbol initially in `fSym`. We modify the `fSym` to use that newly created symbol. Finally, push the global and its local to the parent.

fLocal Simply assign the current procId to the variables declared by the instruction and recursively call the globaliser on the body.

3.1.5.1 Compiler Code Notes

The globaliser is implemented by a function called `Globaliser`. It defines a local class `GlobalsManager` used to ease the management of the newly localised symbols created and their relations to procIds. It also defines a function `AssignScope` used to assign scope to variable declarations. `Globaliser` transforms `fProc`, `fLocal` and `fSym` nodes, and it leaves untouched the nodes `fConst` and `pos`. All other nodes are traversed with the `DefaultPass` function.

3.1.6 CodeGen

The last pass of the compiler will not return a new AST, but a list of opcodes to be passed to the assembler. It is implemented by the function `CodeGen`, which is also responsible for register allocation of symbols present in the AST.

3.1.6.1 Register allocation

Currently, register allocation is very basic as all variable get a Y register assigned. This is far from optimal and offers a big opportunity for improving the performance of the compiled code.

3.1.6.2 Local..in..end

Those Oz instructions are present in the AST as `fLocal` nodes. Generating the opcode for these nodes simply consists in generating the opcodes for the declaration part, followed by the opcodes for the body part of the instruction.

Handling the declaration parts includes assigning a Y register to the symbols, before initialising this Y register with a `createVar` instruction.

Handling the body parts simply consists in traversing the body AST and generate the needed opcodes.

For example, the code fragment in Figure 3.31 declares two variables `A` and `B`, and the opcodes for the declarations is shown in Figure 3.32. These opcodes will be followed by the opcodes for `Body`.

```
local
   A B
in
   Body
end
```

Figure 3.31: Declaration of two variables in Oz

```
allocateY(2)
createVar(y(0))
createVar(y(1))
```

Figure 3.32: Opcodes for the declaration of two variables

3.1.6.3 If then else

Rather than give convoluted explanations, it is easier to look at the code generating the opcodes for `fBoolCase(FSym TrueCode FalseCode Pos)` found in the AST for an `if..then..else` instruction:

```
move({CodeGenInt FSym Params} x(0))|
condBranch(x(0) ElseLabel ErrorLabel)|
%---- true ----
{CodeGenInt TrueCode Params}|
branch(EndLabel)|
%---- error ----
lbl(ErrorLabel)|
move(k(badBooleanInIf) x(0))|
tailCall(k(Exception.raiseError) 1)|
%---- else ----
lbl(ElseLabel)|
case FalseCode
of fNoElse(_) then
   lbl(EndLabel)|nil
else
   {CodeGenInt FalseCode Params}|
   % ---- end ----
   lbl(EndLabel)|nil
end
```

It first moves the symbol containing the boolean to be tested in an X register, which is then used in a `condBranch` opcode, taking as additional arguments the labels identifying the start of the else branch, and the start of the error handling code. The "then" branch directly follows the `condBranch`. After that come the error code and the else branch. Note that at the end of the "then" branch, a branch opcode is needed to jump over the code corresponding to the else branch and the error handling. However, putting the error code before the "else" code avoids an additional jump at the end of the latter. Putting the error code at the end would have required a jump at the end of the "else" code too.

If no code is present in the else branch, no opcode is generated for that branch.

3.1.6.4 Unification with Records

At this step, the only place outside pattern matching (described later) where we can find records are in the right hand side of unifications, i.e. the second features of fEq.

The virtual machine requires, for performance reasons, the compiler to generate different opcodes for cons, tuples and records. If the record is a cons, i.e. a record with label '`|`' and with exactly two features labeled 1 and 2, we need to issue a `createConsUnify(DestReg)`.

If the arity is made of numbers only, i.e. we have the arity of a tuple, we need to issue a `createTupleUnify(k(Label) FeaturesList DestReg)` instruction. Else we need to issue a `createRecordUnify(k(Arity) FeaturesList DestReg)` instruction. The function `CompilerSupport.makeArity` is used to build the arity for use in `createRecordUnify`. It takes as argument the label and a list of

pairs `Label#Value`. If the labels passed in the list are labels of a tuple, it returns false, else it returns the constructed arity. It is based on the value returned by `makeArity` that the compiler decides which instruction is issued. The `FeaturesList` has to be ordered according to the features. This is important, as the `createConsUnify`/`createTupleUnify`/`createRecordUnify` instructions have to be followed by `arrayFill` instructions, one for each value, in the same order as the features passed to the instruction.

As an illustration, this unification of a variable with a record having one value not constant:

```
R=rec(a:A b:2)
```

results in this AST

```
fEq(
   fSym('R' pos)
   fRecord(
      fAtom(rec pos)
      [ fColon( fAtom(a pos)
                fSym('A' pos))
        fColon( fAtom(b pos)
                fInt(2 pos)) ])
   pos)
```

and, if we note `y(Variable)` the register assigned to `Variable` by the compiler, it results in these opcodes being generated:

```
createRecordUnify( k(<Arity rec(a b)>) 2 y(R) )
arrayFill(y(A))
arrayFill(k(2))
```

3.1.6.5 Unification

Unifications involving elementary values (variables or constants) simply issue a `unify(LHS RHS)` instruction, where LHS is the register where the left-hand side is stored and RHS is the register for the right-hand side.

```
A=10
A=B
```

present in the AST in this form:

```
fAnd(
   fEq(
      fSym(Symbol(A) pos)
      fConst(10 pos)
      pos
   )
   fEq(
      fSym(Symbol(A) pos)
      fVar(Symbol(B) pos)
      pos
   )
)
```

where `Symbol(A)` (resp. `Symbol(B)`) represents the instance of the class `Symbol` corresponding to variable `A` (resp. `B`) in this part of the code.

The opcodes generated for this AST are:

```
unify(y(1) k(10))
unify(y(1) y(2))
```

if `A` was allocated register `y(1)` and `B` was allocated register `y(2)`.

3.1.6.6 Procedures/Abstractions

At this stage, only procedures are defined, as found in `fDefineProc` tuples (created by the `Globaliser`, see Section 3.1.5) of the form:

```
fDefineProc(fSym(Sym pos) Args Body Flags Pos NewLocals)
```

As described in Section 2.2.6, the body has first to be assembled. This requires to first assign registers to all symbols used in the body, generating the opcodes for the body (which might itself define procedures and thus do recursive calls) and pass it to the assembler which will return the CodeArea we need. The `createAbstractionUnify` instruction can then be issued, unifying the assembled abstraction's code area `CA` with the register assigned to the symbol `Sym`:

```
createAbstractionUnify(k(CA) GlobalsCount {RegFor Sym})
```

`GlobalsCount` in this instruction is the integer number of global variables present in CA. Each global variable has to be initialised in turn by an `arrayFill` instruction. The first `arrayFill` will initialise the abstraction's `g(0)`, the second will initialise the abstraction's `g(1)`, etc.... For example, if the variable in the enclosing environment referenced by the localised symbol is located in an X register, say 0, then the global is initialised by `arrayFill(x(0))`.

3.1.6.7 Calls

Calls are done with the opcode `call(Callee NumberOfArgs)`. The arguments are not passed to `call`, but will be accessed in registers X with index 0 to `NumberOfArgs`. `CodeGen` will thus first move all argument values to X registers, then issue the `call`. Let's have a look at this example code

```
{F A B 10}
```

and let's suppose that F is located in register y(0), B in y(2) and A in g(2). 10 is accessed via register k(10) as described in Section 2.2.1. The opcodes generated for this call will be

```
move(y(2) x(0))
move(g(2) x(1))
move(k(10) x(2))
call(y(0) 3)
```

3.1.6.8 Case

Case statements are amongst the most complex instructions to handle in code generation. This is because different pattern types are handled differently.

We'll start by ignoring guards for the moment and look at the code in Figure 3.33.

The Mozart1 virtual machine allowed the compiler to group in one test instruction subsequent clauses of constant patterns (records, integers, atoms,

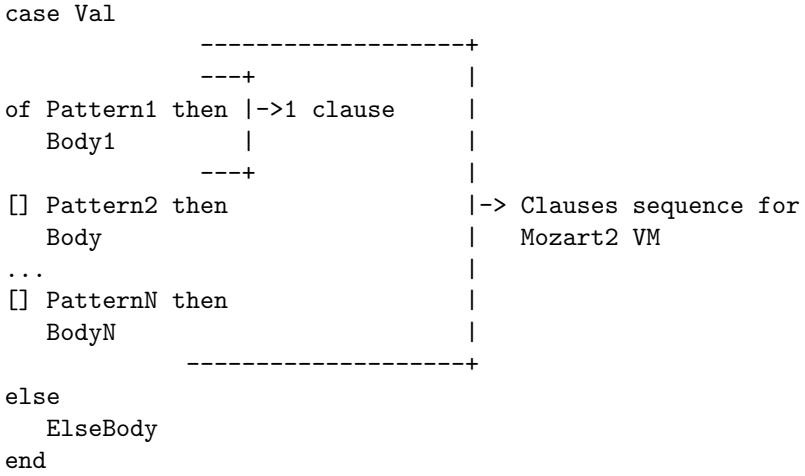

Figure 3.33: Case clauses sequence

floats,...), but open record patterns could only be handled individually. With the Mozart2 virtual machine, both records and open records can be grouped in one test instruction.

We call sequence of clauses a group of subsequent clauses and with no guards. As clauses of a sequence are grouped, the opcode generated will not have one test instruction per clause, but one test instruction per sequence of clauses, and the test result will jump to the corresponding clause' body.

We look here at the code generated for the pattern matching, and don't consider the code declaring the symbols for the captures, as this has been extracted by the namer as described in Section 3.1.2.3.

This will result in opcodes of this form when testing the value in register x(0) against two clauses part of one sequence:

```
patternMatch(x(0) k('#'(Pattern1#lbl1 Pattern2#lbl2)))
branch(ElseLabel)
lbl(1)
... Code for clause1 ...
branch(EndLabel)
lbl(2)
... Code for clause2 ...
lbl(ElseLabel)
... Else Code ...
lbl(EndLabel)
```

The `patternMatch` instruction takes two parameters: the register containing the value to test against, and a constant record with label '#' and with one feature-value pair for each clause (in the order corresponding to the clauses in the Oz code).

The feature is the pattern to which three transformations have been applied.

First capture variables have been replaced by a value resulting from a call to `Boot_CompilerSupport.newPatMatCapture`, passing as unique argument the X

register index in which to store the value of the capture variable in case this pattern matches. The value is the label identifying the start of the code of the corresponding clause, which is where the `patternMatch` instruction will jump in case of a match.

Second, open records have been replaced by an object resulting from a call to `Boot_CompilerSupport.newPatMatOpenRecord`, passing as arguments the arity and features list of the open record.

Third, pattern conjunctions have been replaced by an object resulting from a call to `Boot_CompilerSupport.newPatMatConjunction`, passing as argument the result of the same transformation applied to both features of the `fPatMatConjunction` record.

The patternmatch is immediately followed by an unconditional `branch` instruction, which will jump to the `ElseCode`, but is only reached if no match was found.

When a match is found, the execution jumps to the label indicating the start of the clause' body. The opcodes of the clause' body are prefixed by instructions moving the capture variables'values from their X register to their respectively assigned Y registers. The clause' body opcodes are immediately followed by an unconditional jump to the label indicating the end of opcodes for the case instruction. This ensures that no following clause' body is executed.

Guards, however, add some complexity to the generation of opcodes because clauses with guards cannot be grouped with other clauses. For each clause with guards, there will be a test instruction generated. The code for the guards also needs access to the value of the captures in the pattern, meaning it must come after the test instruction. As a consequence the test instruction may not jump to the start of the clause' body, but must jump to the guards' code. Additionally, when a match is unsuccessful, execution has to jump to the next test.

Here is an example of a case instruction having clauses with guards:

```
case R
of 2 andthen V then
   {Show match}
else
   {Show nomatch}
end
```

and the corresponding opcodes for the case instruction(denoting `y(V)` the Y register allocated for V):

```
1           patternMatch(x(0) k('#'(2#8)))
2           branch(6)
3 lbl(8)    unify(y(1) y(V))
4           move(y(1) x(1))
5           condBranch(x(1) 6 4)
6           move(k(match) x(0))
7           call(k(<P/1 Show>) 1)
8           branch(3)
9 lbl(4)    move(k(errorInCase) x(0))
10          tailCall(k(<P/1 Exception.raiseError>) 1)
11 lbl(6)   move(k(nomatch) x(0))
12          call(k(<P/1 Show>) 1)
13 lbl(3)   return
```

The `patternMatch` instruction is built the same way, but it now jumps to the guards' code at `lbl(8)` in case of success.

The guards' code, which here only consists of line 3, is immediately followed by the code of a conditional jump (lines 4 and 5) to the next test or, if as here it is the guards of the last clause, to the else code (`lbl(6)`). The `condBranch` also takes a label for error code, which in this case is label `4`.

The code of the clause is on line 6 and 7, followed by the unconditional jump to the end label, here `lbl(3)`.

As clauses with guards can be mixed with clauses without guards, the opcodes generated will contain some test instructions covering multiple clauses while some tests will cover only one clause.

3.1.6.9 Instructions Sequences

Generating opcodes for sequences of instructions consists in the generation in order of the opcodes for each element of the sequence. When `CodeGen` visits a node `fAnd(First Second)` in the AST, it first generates the opcodes for `First`, then for `Second`.

3.1.6.10 Exceptions

The nodes present for the `try` statements in the AST handled by the code generator are of the form

```
fTry(Body
     fCatch( [fCaseClause(E Case)]
             CatchPos)
     fNoFinally
     Pos)
```

It corresponds to the code

```
try Body catch E then Case end
```

and translates in opcodes as:

```
setupExceptionHandler(TryLabel)
{CodeGen Case}
branch(EndLabel)
lbl(TryLabel)
{CodeGen Body}
popExceptionHandler
lbl(EndLabel)
```

`setupExceptionHandler(TryLabel)` sets up an exception handler, and then jumps to the `TryLabel`, which is the location of the opcodes for `Body`. If an exception is raised, the execution jumps to the instruction following the setup of the handler. If the `Body` executes successfully (without exception), the exception handler is removed, and execution continues after the `EndLabel`.

Notice that when an exception is raised, the exception handler jumps in, executes its code, and then jumps to the end label over the `popExceptionHandler`. This has to be so because the virtual machine removes the exception handler when it treats the exception.

3.1.6.11 Compiler Code Notes

The functions `CodeGen` declares multiple local functions:

- `CodeGenDecls` will generate opcodes for symbols in declaration parts. This currently consists in assigning a yindex to the symbol and issuing a `createVar(y(yindex))` instruction.
- `RegForSym` will generate opcode to access the register of the symbol passed as argument
- `PermRegForSym` will generate opcode to access the Y register of the symbol passed as argument. This is needed to specifically access the Y register of a symbol for a pattern matching capture, which also has an X register assigned.

Most operations applied to AST nodes in `CodeGen` are straight-forward translation of the descriptions given in this text to Oz code. For example, here is the code handling fLocal nodes in the AST:

```
%---------------------
of fLocal(Decls Body _) then
%---------------------
   [ {CodeGenDecls Decls Params} {CodeGenInt Body Params}]
```

It just generates opcodes for declarations followed by the opcodes of the body of the `local..in..end` Oz instruction, as described in Section 3.1.6.2.

However, due to the higher complexity of the code handling `fCase` nodes, we'll spend some time analysing it here.

As a reminder, we call sequence a group of consecutive clauses that can be grouped in one test instruction in the opcodes generated. As clauses with guards cannot be grouped with other clauses in one test instruction, these can only be part of a sequence of which they are the only clause!

We call prefix of a sequence the test instructions checking the pattern and the guards before jumping to the clause' code, the next test instruction, the else code, or the end label. The prefix also includes the code moving captures, available in X registers, to Y registers if needed. The prefix of a sequence is generated by a call to the function `PrefixOfSeq` which itself calls `UsedSymbolsToYReg` to generate opcode to move symbols used in a pattern to Y registers.

`CodeGen` visits all clauses in turn.

As the record passed to the instruction `patternMatch` needs to include the label of the jump destination in case of a match, opcodes for all clauses have to be generated before the prefix can be generated.

This is why the code uses a variable `Code`, containing code, including prefix, for all completely visited sequences; and a variable `CodeBuffer`, containing code for all visited clauses of the sequence currently visited. When a new sequence is started, the prefix of that sequence and its `CodeBuffer` are appended to `Code`, and `CodeBuffer` is reset.

As all clauses except those with guards can be grouped, a clause starts a new sequence in two cases:

- the clause has guards
- the clause immediately follows a clause with guards

For this, the compiler code defines the function `IsNewSequence` which returns true if the clause passed as argument is starting a new sequence. The function needs to get passed as arguments the current clause and the current sequence type, held in the variable `SequenceType`

When a new sequence is started, the previous sequence has to be closed: its prefix and code buffer must be added to `Code`. However, building the prefix for a clause with guards requires knowledge of the register in which the value of the guards expression is stored. This is found in the fourth features of the pattern's `fNamedSideCondition` tuple in the AST, and is why `fNamedSideCondition` has an additional feature compared to `fSideCondition`. The last visited clause' pattern is stored in `ThisPattern`. This is needed for the closing code, which also generates the opcode for the last sequence.

As described, the test instruction `patternMatch` has to jump to the clause' body code or the guards code, identified by the label in `ThisLabel`. If there is no match, or if there is a match, but the guards are not respected, the code must jump to the next test, identified by the label in `NextTestLabel` This means that when closing the current sequence, its test instruction has to be identified by a label, which is found in `ThisTestLabel`.

The record to be passed to the instruction `patternMatch` is reset to `'#'()` when a new sequence is started, and is updated for each clause of the sequence. The record built with all visited clauses of the current sequence is in the variable `PatternMatchRecord`. Each clause updates its respective feature value in `PatternMatchRecord` through a call to the function `TransformPattern`. A clause' feature number is tracked by the variable `SeqLen`, holding the number of clauses already visited in the current sequence. As all capture values have to be available to the clause' code, symbols encountered in the pattern are collected in `UsedSymbols` by `TransformPattern`. It is also `TransformPattern` that assigns an X register to each capture. The last X register that has been assigned is kept in `XIndex`

Now that we have all elements of the code defined, we can define the invariant of the loop over the clauses:

- `SequenceType` is the type of the currently visited sequence
- `ThisTestLabel` is the label identifying the test instruction of this sequence
- `NextTestLabel` is the label that will identify the test instruction of the next sequence
- `Code` is the list of opcodes for all completely visited sequences
- `CodeBuffer` is the code for all visited clauses of the current sequence
- `PatternMatchRecord` is the record to be passed to `patternMatch` holding information for all visited clauses of the current sequence.
- `SeqLen` is the number or clauses already visited in the current sequence
- `LastPattern` is the pattern of the last visited clause.

`XIndex`, `ThisLabel` and `UsedSymbols` are not included in the invariant as these are only used inside each iteration of the loop and reset at the beginning of each clause handling.

With this invariant in place, we can sketch the structure of the code in Figure 3.34.

```
{List.forAllInd Clauses proc{$ Ind fCaseClause(Pattern Body)}
   if {IsNewSequence {Label Pattern} @SequenceType} then
      % close previous sequence
      % Ensures Code respects the invariant at the end of this iteration
      --- code not included ---
      % Reset sequence variables,
      % This ensures that SequenceType, ThisTestLabel, NextTestLabel
      % respect the invariant at the end of this iteration
      CodeBuffer:=nil SeqLen:=0
      PatternMatchRecord:='#'()
      SequenceType:={Label Pattern}
      ThisTestLabel:=@NextTestLabel
      NextTestLabel:={GenLabel}
   end
   % Reset clause specific variables
   ThisLabel:={GenLabel} UsedSymbols:=nil XIndex={NewCell 0}

   % Build record used in the pattern matching instruction
   % Ensures PatternMatchRecord respects the invariant at the end
   % of this iteration
   PatternForRecord = {TransformPattern Pattern XIndex UsedSymbols}
   PatternIndex=@SeqLen+1
   PatternMatchRecord:={Record.adjoin
                        @PatternMatchRecord
                        '#'(PatternIndex:PatternForRecord#@ThisLabel)}

   % Update CodeBuffer to respect invariant
    --- code not included ---
   % Restore invariant
   SeqLen:=@SeqLen+1
   LastPattern:=Pattern
end}
% Add last clause, error handling code, the else code and the
% label identifying the end of the case instruction's code
Code:=@Code|
      PrefixOfLastSequence
      @CodeBuffer|
      lbl(ErrorLabel)|        %---- error ----
      move(k(errorInCase) x(0))|
      tailCall(k(Exception.raiseError) 1)|
      lbl(@NextTestLabel)|    %---- else ----
      case Else
      of fNoElse(_) then
         lbl(EndLabel)|nil
      else
         {CodeGenInt Else Params}|
         lbl(EndLabel)|nil
      end
```

Figure 3.34: Case loop structure

3.2 Compiling to a file

The functor in 3.23 is clearly an expression, but the virtual machine can only execute statements. the compiler applies a trick. When compiling an expression, the compiler unifies the expression with a variable `Result` it declares, and it then executes that statement.

The current implementation uses a constant function `BindResult` listed in Figure 3.35, which is injected in the AST received from the parser as illustrated in Figure 3.36. This modified AST is compiled and executed, and the value of `Result` is then written to the output file with Pickle.

```
proc {BindResult Value}
   Result = Value
end
```

Figure 3.35: `BindResult` procedure

```
fApply(fConst(BindResult pos) [ AST ] pos)
```

Figure 3.36: `BindResult` call injected in the AST

3.3 Tests

3.3.1 Helper functions tests

Some helper functions are tested specifically by Oz code, checking that the result returned by the function is correct. This is simply done by calling the function in Figure 3.37 with the first argument the function call we want to test, and the expected result as second argument as illustrated in Figure 3.38 where the function UnWrapFAnd is tested. The only downside to this approach is that helper functions have to be exported for them to be available in the test code.

```
proc {Equals Result Expected}
   if Result\=Expected then
      %Show error information
      raise unexpectedResult end
   end
   {System.printInfo '.'}
end
```

Figure 3.37: HelpersTests function

```
{Equals
   {Compile.unWrapFAnd fAnd(first fAnd(second third) )}
   first|second|third|nil }
```

Figure 3.38: Helper test example

3.3.2 Compiler tests

Tests are defined by three files each:

- the oz code to compile and execute
- the expected standard output
- the expected standard error

All these files are put under the tests/definitions directory.

A `TestRunner` script in oz loads the code found in the file whose path is passed as argument, parses it, gives the AST to the compiler which returns the opcodes which are assembled and executed. A shell script iterates over all tests and for each execute the `TestRunner`, puts the standard output and standard error in result files under the tests/results directory, compares their content with the expected results and in case of difference can open a diff-viewer (like vimdiff). All test code has a preamble comment describing what case this test covers. If the test code contains the comment line

%-- SKIP TEST--

the test is skipped. This was implemented to be able to run tests even if one of them was temporarily not passing.

Adding a test is very simple:

1. put the code to compile and execute in an .oz file under tests/descriptions
2. put the expected output in a file with same name but with extension .out
3. put the expected error output in a file with same name but with extension .err

Once this is done, the new test will be included in the next run.

All tests are run with

```
make tests
```

It is also possible to run one individual test with

```
make test test=$testnumber
```

In this case, the test is run even if it is marked as to be skipped when running the full test suite. This helps writing tests for cases needed to be supported in the future, but that should not make the full test suite fail.

As the virtual machine is still in development, it is possible to restart a tests run from a specific test. This avoids to re-run all passed tests in case of a hanging run:

```
make testsfrom from=$testnumber
```

3.4 Performance

This section will compare the execution time of source code

- compiled by the Mozart1 compiler and running on the Mozart1 virtual machine
- compiled by the adapted Mozart1 compiler targeting the new Mozart2 virtual machine
- compiled by the compiler described in this document, running on the new Mozart2 virtual machine

To obtain relevant results, comparable measures needed to be collected under the three circumstances. To avoid garbage collection penalties, the garbage collector is called explicitly before the first measure of time with `{System.gcDo}`. Code was structured as to avoid the static analysis of the Mozart1 compiler to optimise the code aggressively (for example in the `case` measures), but still one test was not possible: the Mozart1 compiler discarded constant record unification with a wildcard.

For the code running on the Mozart1 virtual machine, the timing is done by calling

```
{Property.get 'time.total'}
```

before and after the call to the code of which we want to measure the execution time, and computing the difference. The resolution obtained for these measure was 10ms.

For the code running on the Mozart2 virtual machine, the timing is done the same way by calling

```
{Boot_Time.getReferenceTime}
```

The resolution of the measures was 1ms. In both cases, the result obtained is the total running time of the code in milliseconds.

Most measures were done by executing 100 times one operation in a loop doing 10^5 iterations. The cost of the loop was evaluated by measuring the time taken by 10^7 iteration of an empty loop, and bringing it back to 10^5 iterations, to limit the imprecision due to the resolution of 10ms of the measures on Mozart1. The average cost of the loop doing 10^5 iterations is about 4ms, 1.9ms and 0.23ms respectively for the compiler described in this document, the Mozart1 compiler targeting the Mozart2 virtual machine, and the Mozart1 compiler targeting the Mozart1 virtual machine. These times are negligible compared to the total execution times measured, being for example less than 0.4% in the worst case of the `Addition` performance test.

Each measure is done 20 times, and the average and standard deviation is computed. Results can be found in F.

In all measures, the code running on the Mozart1 virtual machine was the fastest, followed by the Mozart1 compiler targeting the Mozart2 virtual machine, and with the new compiler being the slowest. This is normal as the focus of the implementation was producing clean and modular code. The `CodeGen` function is currently very naive, but provides a foundation on which to implement state of the art optimisations.

We will make two comparisons:

- the running time of the code generated by the Mozart1 compiler running on the Mozart2 virtual machine to the running time of the code generated by the Mozart1 compiler running on the Mozart1 compiler.
- the running time of the code generated by the new compiler running on the Mozart2 virtual machine to the running time of the code generated by the Mozart1 compiler running on the Mozart2 compiler.

Two reasons for this:

- we compare a measure to the first faster alternative
- measures compared have at least on element in common: the compiler of the virtual machine

A simple addition was run by the function listed in Figure 3.39.

```
proc {Addition N}
   for I in 1..N do
      _ = 1+1  % line repeated 100 times
      ...
   end
end
```

Figure 3.39: Simple addition

The function was called with 10^5 as argument. The Mozart1 compiler targeting the Mozart2 virtual machine was 10 times slower than the Mozart1 setup. The new compiler was more than 4 times slower than the Mozart1 compiler targeting the Mozart2 virtual machine.

A naive Fibonacci function displayed in Figure 3.40 was tested.

```
  fun{SlowFib N}
   if N==0 then 0
   elseif N==1 then 1
   else
      {SlowFib N-2}+{SlowFib N-1}
   end
end
```

Figure 3.40: Naive Fibonacci implementation

The function was called with `30` as argument. The Mozart1 compiler targeting the Mozart2 virtual machine was more than 5 times slower than the Mozart1 setup. The compiler described in this document was more than 2 times slower than the Mozart1 compiler targeting the Mozart2 virtual machine. For further tests we will note this as a slowdown ratio of 5 and 2.

Code writing to cells was tested with the function in Figure 3.41.

The slowdown ratios were 2.68 and 3.01.

Code reading cell values was tested with the function in Figure 3.42.

```
proc {CellWrite N}
   C={NewCell 0}
in
   for I in 1..N do
      C:=1 %100 times
      ...
   end
end
```

Figure 3.41: Writing to cells

```
proc {CellRead N}
   C={NewCell 0}
in
   for I in 1..N do
      _=@C %100 times
      ...
   end
end
```

Figure 3.42: Code reading cell value

The slowdown ratios were 6.27 and 3.84. The code generated by the compiler described in this document is slower to read from cells than to write to them. This is because the code generated for reading a cell is not optimal and contains a unification, as illustrated in Figure 3.43, which has a high cost in terms of time. Writing to a cell does not generate a `unify` operation.

```
createVar(y(1))
createVar(y(2))
unify(y(1) y(2))
moveMove(y(0) x(0) y(1) x(1))
tailCall(k(<P/2 Value.catAccess>) 2)
```

Figure 3.43: Opcodes for reading a cell

Performance of code using cells in both read and write operations was also measured, with the code displayed in Figure 3.44.

The function was called with 10^5 as argument. The slowdown ratio was 2.1 and 2.8.

As displayed in Figure 3.45, the new compiler generate code that is 2 to 3 times slower in average than the code generated by the Mozart1 compiler targeting the Mozart2 virtual machine. There is one kind of code for which it can approach the performance of the old compiler though: pattern matching. Because the new compiler combines constant patterns and open record patterns clauses without guards in one `patternMatch` instruction (see Section 3.1.6.8), it can generate code that is more efficient for the pattern matching(though it stays slower due to other parts of the code being more inefficient). In the favorable

```
proc {CellAccess N}
   C={NewCell 0}
in
   for I in 1..N do
      C:=@C+I % 100 times
      ...
   end
end
```

Figure 3.44: Cell access testing function

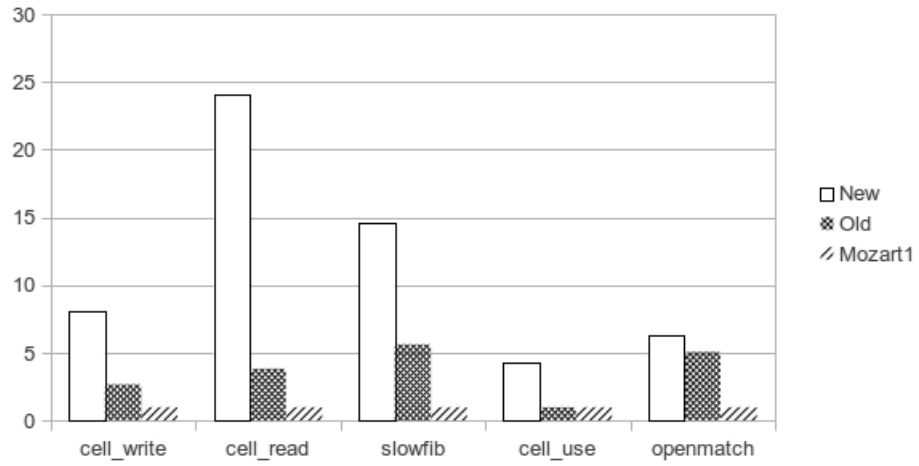

Figure 3.45: Speed of generated code relative to the Mozart1 platform. Smaller is better.

case where it is the last clause of a long list of open record pattern clauses that matches, like the function in Figure 3.46 called with `A=rec(f:1 z:3 y:4 x:6)` and `N=`10^5 the new compiler generates code that is only 24% slower than the code generated by the Mozart1 compiler.

```
proc {OpenMatch A N}
   for I in 1..N do
      case A     % 100 times
      of 1 then
         skip
      [] rec(a:_ ...) then
         skip
      [] rec(b:_ ...) then
         skip
      [] rec(c:_ ...) then
         skip
      [] rec(d:_ ...) then
         skip
      [] rec(e:_ ...) then
         skip
      [] rec(f:_ ...) then
         skip
      [] 2 then
         skip
      end
   end
end
```

Figure 3.46: Function to test `case` performance

Chapter 4

Conclusion

4.1 Achievements

Most of the goals set when this work was started have been reached. Although the whole Oz language is not supported (see the next section), a vast majority is, as is demonstrated by the fact that all programs extracted from [VRH04] compile and run successfully. The code generated by the compiler, although not optimised, is exploiting the capabilities of the new virtual machine, notably in pattern matching. During the whole development, attention has been paid to produce code easy to understand and fully documented, which should make it easy for new developers to dive into the code.

4.2 Future work

4.2.1 Missing language support

Support for these Oz language features have to be added:

- for loops iterating over multiple lists
- C-like for loops
- record pattern arguments having non-constant features

Support for logic programming features of Oz also have to be added.

4.2.2 A better try-finally transformation

A more efficient transformation can be implemented for `try-finally` as illustrated in Figure 4.1 and Figure 4.2.

For an expression, the result of the transformation should be as illustrated in Figure 4.3 This transformation takes advantage of the fact that if the variable waited for gets bound to a failed value, `Wait` will re-raise its encapsulated exception. It is more efficient than the current implementation presented in Section 3.1.3.15 because there's no pattern matching involved, there's no wrapping of the value of the expression if the code executes successfully, and the test is done in C++.

```
try <Body> finally <Finally> end
```

Figure 4.1: Try-finally instruction

```
local Test in
   try
      <Body>
      Test = unit
   catch E then
      Test = {Value.failed E}
   end
   <Finally>
   {Wait Test}
end
```

Figure 4.2: New transformation for Figure 4.1

```
local Y Test in
   try
      Y = <Body>
      Test = unit
   catch E then
      Test = {Value.failed E}
   end
   <Finally>
   {Wait Test}
   Y
end
```

Figure 4.3: New transformation for a try expression

4.2.3 Better calls to builtins

The call to builtins can also be improved. Currently, every call is translated in a `call` opcode. Calls to builtins should be translated by the opcode `callBuiltin`. Determining if the callee is a builtin can be done with

```
{CompilerSupport.isBuiltin Callee}
```

Information about the builtin's parameters can be obtained by a call

```
BuiltinInfo = {CompilerSupport.getBuiltinInfo Callee}
```

yielding a record of the form

```
builtin(arity:N name:Name params:Params ...)
```

where `Params` is a list of `param(kind:InOut ...)`, `InOut` having value `in` for input parameters, `out` for output parameters. `N` is the number of parameters if the builtin, and should be used to validate the number of arguments found in the call. If all is correct, opcodes can be generated. The `in` parameters are `moved` to X registers before the `callBuiltin`, and out parameters are extracted by unification of their register with the result register. Here is an example for the builtin `Number.'+'`, taking 2 parameters `in` and one `out`. This AST for the call

```
fApply( fConst(Number.'+')
        [y(3) y(1) y(6)]
        pos)
```

should result in these opcodes:

```
move(y(3) x(0))
move(y(1) x(1))
callBuiltin(k(Callee) [x(0) x(1) x(2)])
unify(x(2) y(6))
```

4.2.4 Improve CodeGen

Another area of improvement is the `CodeGen` function, both its inner working and its output should be improved. Its inner working because it currently generates nested lists that needs to be flattened at the end. Its output can greatly be improved, notably in the register allocation, which is currently not optimised. The latter should improve the performance of the generated code.

4.2.5 Others

Further limiting the use of cells and striving for a totally functional code-base should also help performance and modularity.

Furthermore, integration in the Mozart development environment also has to be started and some debug output is still generated.

Bibliography

[OzOv] http://en.wikipedia.org/wiki/Oz_(programming_language)

[Moz] http://en.wikipedia.org/wiki/Mozart_Programming_System

[VRH04] Peter Van Roy and Seif Haridi. Concepts, Techniques, and Models of Computer Programming. MIT Press, 2004.

[Moz1.4] Mozart Programming System, version 1.4.0 (www.mozart-oz.org).

[Aho07] Aho, A. (2007). Compilers : principles, techniques, & tools. Boston: Pearson/Addison Wesley

[BaseLang] Oz - The Base Language
http://www.mozart-oz.org/documentation/notation/node6.html

[AST] Oz - Syntax Tree Format
http://www.mozart-oz.org/home/doc/compiler/node7.html#appendix.syntax

[CTMCPSup] Supplements for "Concepts, Techniques, and Models of Computer Programming"
http://www.info.ucl.ac.be/ pvr/ds/mitbook.html

[BootComp] Mozart2 BootCompiler by Sébastien Doeraene
https://github.com/mozart/mozart2-bootcompiler/tree/master

[Moz2vmsrc] Mozart2 virtual machine
https://github.com/mozart/mozart2-vm

[Moz2src] Mozart2 source code
https://github.com/mozart/mozart2

[Func] More on Oz Functors
http://www.mozart-oz.org/home/doc/apptut/node6.html#chapter.mof

[FuncCode] Oz Application Development - Functors (http://www.mozart-oz.org/home/doc/apptut/node3.html#section.development.functors)

[CTMCPFigs] Figures of "Concepts, Techniques, and Models of Computer Programming" (http://www.info.ucl.ac.be/ pvr/bookfigures/)

Appendix A

Pattern Variables

The Pattern Variables are the variables declared by an instruction placed in the declaration part of a `local..in..end` instruction.

A.1 Statements

`D1 D2`	`PV(D1)` $\cup$ `PV(D2)`
`x`	`{x}`
`(S)`	`PV(S)`
`(D in S)`	`PV(S) - PV(D)`
`local D in S end`	`PV(S) - PV(D)`
`proc ... {E ...} ... end`	`PV(E)`
`fun ... {E ...} ... end`	`PV(E)`
`class E ... end`	`PV(E)`
`functor E ... end`	`PV(E)`
`E = ...`	`PV(E)`
`otherwise`	$\emptyset$

A.2 Expressions

`x`	`{x}`
`(E)`	`PV(E)`
`(D in [ S ] E)`	`(PV(S)`$\cup$`PV(E)) - PV(D)`
`local D in [ S ] E end`	`(PV(S)`$\cup$`PV(E)) - PV(D)`
`E1 = E2`	`PV(E1)`$\cup$`PV(E2)`
`[E1 ... En]`	`PV(E1)`$\cup$ `...` $\cup$ `PV(En)`
`E1|E2`	`PV(E1)`$\cup$`PV(E2)`
`E1#...#En`	`PV(E1)`$\cup$ `...` $\cup$ `PV(En)`
`l([ f1: ] E1 ... [ fn: ] En [ ... ])`	`PV(E1)`$\cup$ `...` $\cup$ `PV(En)`
`otherwise`	$\emptyset$

Appendix B

OpCodes

This table lists the opcodes supported by the assembler.

Opcode	Arguments	Notes
`skip`		No-op, does nothing
`move(r s)`	`r`: register X, Y, G or K `s`: register X or Y	Copies the value of `r(i)` in `s(j)`.
`moveMove(r s t u)`	`r`, `t`: register X, Y, G or K `s`, `u`: register X or Y	Automatically generated by the assembler when encountering two consecutive `move` operations. Equivalent to `move(r s) move(t u)`.
`allocateY(N)`	`N`: integer>0	allocates `N` Y registers in current frame. Can only be called once per frame.
`createVar(r)`	`r`: register X or Y	Assigns a new unbound variable to register `r`
`createVarMove(r x)`	`r`: register X or Y	equivalent to `createVar(r) move(r x)`
`setupExceptionHandler(l)`	`l`: label	sets up an exception handler, and execution jumps to label `l`
`popExceptionHandler()`		removes the exception handler currently set up. Should not be called when an exception was raised, as the virtual machine pops it in that case

`callBuiltin(k(BI) Args)`
`callBuiltin0(k(BI))`
`callBuiltin1(k(BI) x1))`
..
`callBuiltin5(k(BI) x1 .. x5)`
`callBuiltinN(k(BI) N x1 .. xN)`
	builtin in a K register, all others are X registers	The assembler produces the right specialised version from the generic `callBuiltin`

Opcode	Arguments	Notes
`call(r N)` `tailCall(r N)`	`r`: register X, Y, G, K `N`: number of arguments of the call	The `N` arguments need to be passed in registers X from `x(0)` to `x(N-1)`
`sendMsg(r k(L) N)` `tailSendMsg(r k(L) N)`	`r`: register X, Y, G, K `L`: record label / arity `N`: number of features of message / number of features of arity	Equivalent to `{r M}` in Oz notation, where `M` is a tuple (or a record of arity L) of label `L` with `N` features, whose values are found in registers `x(0)` to `x(N-1)`. Optimises method call on objects by avoiding building the message tuple then issuing a `call`
`return()`		Frees Y registers, pops the top frame from the stack, and continues execution based on the new top frame.
`branch(L)`	`L`: label	moves execution to the label `L`
`condBranch(x LF LE)`	`LF`: label identifying the "false" branch `LE`: label identifying the error handling code	if `x==true` then does nothing. if `x==false` then jumps to label `LF`. In other cases, jumps to label `LE`.
`patternMatch(r k(H))`	`r`: register X, Y, G `H`: see Section 3.1.6.8	
`unify(r s)`	`r` and `s`: register X, Y, G, K	Unifies the content of registers `r` and `s`
`createAbstractionUnify(k(CA) N r)`	`CA`: CodeArea `N`: number of globals `r`: register X, Y, G, K	Must be followed by `N` `arrayFill` instructions to initialise the globals
`createAbstractionStore(k(CA) N r)`	`CA`: CodeArea `N`: number of globals `r`: register X, Y	Must be followed by `N` `arrayFill` instructions to initialise the globals
`createRecordUnify(k(A) N r)`	`A`: Arity, i.e. label and features of the record `N`: number of features `r`: register X, Y, G, K	Must be followed by `N` calls to `arrayFill` to set the value corresponding to each feature, in the same order as the features in Arity.
`createRecordStore(k(A) N r)`	`A`: Arity, i.e. label and features of the record `N`: number of features `r`: register X, Y	Must be followed by `N` calls to `arrayFill` to set the value corresponding to each feature, in the same order as the features in Arity.
`createTupleUnify(k(L) N r)`	`L`: label of the tuple `N`: number of features `r`: register X, Y, G, K	Must be followed by `N` calls to `arrayFill` to set values corresponding to each feature. Features of a tuple are all subsequent integer from 1.
`createTupleStore(k(L) N r)`	`L`: label of the tuple `N`: number of features `r`: register X, Y	Must be followed by `N` calls to `arrayFill` to set values corresponding to each feature. Features of a tuple are all subsequent integer from 1.

Opcode	Arguments	Notes
`createConsUnify(r)`	`r`: register X, Y, G, K	Must be followed by two calls `arrayFill` to set the values of features `1` and `2`, the only two features of a cons.
`createConsStore(r)`	`r`: register X, Y	Must be followed by two calls `arrayFill` to set the values of features `1` and `2`, the only two features of a cons.
`arrayFill(r)`	`r`:register X, Y, G, K	the ith call to `arrayFill` will fill the ith placeholder of the preceding `create...` instruction
`arrayFillNewVar(r)`	`r`: register X,Y	creates a new unbound variable in `r` and puts it in the placeholder corresponding to the `arrayFill` call.
`arrayFillNewVars(N)`	`N`: integer >0	equivalent to `N` subsequent calls to `arrayFill`, putting an distinct unbound variable in each placeholder

Appendix C

Symbol Description

A symbol instance holds information about the variable it replaced in the AST. Symbols introduced by the compiler are called synthetic symbols. The attributes of the class `Symbol` are:

id A unique id for the symbol, used for debugging purposes.

name The name of the variable it represents. This can be empty for symbols introduced by the compiler.

pos The position of the variable's declaration in the source code.

xindex yindex gindex The indexes of the registers assigned to the symbol. A symbol might have both an xindex and a yindex when it is a capture in a pattern. The xindex is the index in which the value of the capture is placed, and the yindex is the index of the Y register assigned to the symbol for permanent storage in the clause' body.

procId The procId in which this symbol is reachable.

type The type of the symbol, which can take the following values:

- localProcId for a symbol corresponding to a variable defined in the procedure where it is located.
- localised for a symbol representing a global variable for the procId where it is located
- patternmatch for a symbol representing a capture in a pattern
- wildcard for a symbol introduced in the place of a wildcard

ref is only used for localised symbols. `ref` references a symbol local to the enclosing procedure, building a chain that will eventually lead to a symbol of type localProcId in the procedure declaring the variable, enabling the resolution of global variables.

Appendix D

Tests list

Here is a description extracted from the tests written (`make desc` generates it dynamically). The suite consists of 439 test scripts, performing more than 1000 output checks.

```
001 nested locals:
 Nested locals, with one unification done in the declaration of B.
002 nested locals:
 The top local declared and initialises variables A and B.
 The nested local redeclares B and initialises it with another value.
 In the nested local, we then show both A and B.
 In the outer local, we show A and B to check it has the top level value.
003 nested locals:
 Three levels deep nested locals, with redeclaration and new variable at each
 level.
005 declarations:
 Declarations in records (PVS and PVE)
015 procs:
 Wildcard in formal parameters
020 nested procs:
 Nested proc accessing var 1 and 2 levels higher
021 nested procs:
 Global in proc
022 nested procs:
 Use of a variable after nested proc definition using it also
023 nested procs:
 Variable used before a nested proc also uses it
024 nested proc siblings:
 Two nested proc definitions (T and W)  at the same level use the same variable
 (A) which is also a new local to the outer proc (P), which is used after the
 nested procs definitions.
025 nested proc siblings:
 Two nested proc definitions (T and W)  at the same level use the same variable
 (A) which is also a new local to the outer proc (P), which is used *before*
 the nested procs definitions.
026 nested proc siblings:
 Two nested proc definitions (T and W)  at the same level use the same variable
```

```
 (A), but T redeclares it and W uses the global. The outer proc uses
 A before its children proc definitions.
027 nested proc siblings:
 Two nested proc definitions (T and W)  at the same level use the same variable
 (A), but T redeclares it and W uses the global. The outer proc uses A *after*
 its children proc definitions.
028 nested proc:
 Locally vars overriding globals
029 globaliser:
 Is environment restored avec proc definition?
050 desugar unnester:
 Simple desugar of + - * / and unnesting of that operation's result assignment
 Uses a negative number too
051 desugar unnester:
 Simple desugar a function
052 desugar unnester:
 Desugar of functions returning functions.
 All called functions return a function without argument that itself has to be
 called, hence the double {{ }}
053 desugar unnester:
 Cell creation, assignment, access, exchange, cell in cell
054 desugar unnester:
 Wildcard   for argument set by procedure.
055 desugar unnester:
 Functional procedure definitions (nesting marker in arguments list)
056 desugar unnester:
 Unification expressions
060 unnester:
 Unnesting of variable assignment in declarations of a local and in proc
 argument
061 unnester:
 Unnesting of variable assignment in declarations of a local and in proc
 argument
062 unnester:
 Unnesting of variable assignment with variable being the RHS and in proc
 argument
063 unnester:
 Unnesting of variable assignment with variable being the RHS and in proc
 argument
064 unnester:
 Unnesting of locals in unification
065 unnester:
 Unnesting of locals in unification and arithmetic
066 unnester:
 Unnesting of binding with LHS and RHS both non-elementary
067 unnester:
 Unnest the taget of a call
068 unnester:
 Dollar present in a record argument of a procedure
069 unnester:
```

```
 Unnest case values
100 if:
 Simple if statement tests
101 if:
 Complex if statements and expressions, with condition and branches
 non-elementary. Also tests comparators < =< > >=
102 if:
 Test if non-boolean are treated correctly
103 if:
 Elseif test.
105 if:
 If statements with no else
110 lists:
 Simple access to list elements
130 for:
 Loop 'for' over lists
131 for:
 Statement 'for' over lists with pattern matching
132 for:
 Statement 'for' over integers
133 for:
 For loop over multiple lists
 Skipping until I fix the multiple list behaviour
150 records:
 Simple record creation and access
151 records:
 Auto number features of records if necessary (desugar step)
152 records:
 Constant records
153 records:
 Records where values are not constants
154 records:
 Records where values are results of function call
155 records:
 Records where the label is not constant
156 records:
 Records with all parts (label, features, values) specified by a the value of a
 variable.
157 records:
 Records on LHS of unification, both const and not const
158 records:
 Record with value result of an if expression
159 records:
 Unification of record to assign value by 'pattern matching'
160 records:
 Create Cons
180 nesting marker:
 Nesting marker in proc argument, and in if statement
190 dotassign:
 Test dotAssign expressions
```

```
200 threads:
 Unification of a variable in a thread, depending on variables initialised
 later.
201 threads:
 Thread as expression, at the 2 sides of unification
202 threads:
 Function calls in thread expressions
203 threads:
 Wait instruction, waiting for variable bound in thread.
220 lazy functions:
 Lazy functions. Result variables used in opposite order than unification, and
 one result unused.
230 streams:
 List with unbounded tail filled at one second interval
231 streams:
 List with unbounded tail filled at one second interval
250 locks:
 Locks
260 case:
 Records and OpenRecord matches
261 case:
 Case with missing else
262 case:
 Case expression
263 case:
 Case expression with no else
264 case:
 Case with only constant tests, i.e. no capture
265 case:
 Cases with captures and guards
266 case:
 Wildcards in record and openrecords patterns
267 case:
 Test !Vars in patterns
268 case:
 Pattern conjunction
269 case:
 Pattern matching in function and proc arguments
270 case:
 Pattern arguments, proc and fun
271 case:
 Pattern conjunctions open records and escaped variables in function arguments
272 case:
 Parameter arguments test for open record with atom features and not integers
300 classes:
 Simple class instanciation and object method call
301 class:
 Key arguments in class methods
302 classes:
 Default values for method arguments
```

```
303 classes:
 Attribute access, including with cell values + accessing non-attribute cells
 in a method.  Also access attribute which name is accessed via a variable.
304 classes:
 Exchange operation on attribute, and assigning to an attribute with no default
 value.
305 classes:
 Class feature access
306 classes:
 Method with no argument and class with not feat or attr.
307 classes:
 Simple class inheritance
308 classes:
 Inheritance and features
309 classes:
 Wildcard in feature
310 classes:
 Anonymous class
311 classes:
 Call a method on self
312 classes:
 Attribute assignation with <-
313 classes:
 Static calls
314 classes:
 Method calls as expression
315 classes:
 Static call with nesting marker
316 classes:
 Functional methods
317 classes:
 Test of pattern variable function for classes
318 classes:
Generic classes
319 classes:
 Open record method definitions
320 classes:
 Private and dynamic method labels
 also tests otherwise method
321 classes:
 Private method which has same name has its containing class
322 classes:
 Method head reference
323 classes:
 Method head captures with same name
324 classes:
 Dynamic attribute access
325 classes:
 Private method with same variable name as class
326 classes:
```

```
 Method head capture for method with no argument
327 classes:
 Locking property
328 classes:
 Inherintance from a final class
350 raise:
 Exception raised
370 try:
 Simple try catch statement
371 try:
 Simple try catch expression
372 try:
 Corner case illustrating the need of temp var when desugaring try expression
373 try:
 try..finally statement
 Exception raised
374 try:
 try-finally expressions
 Exception raised
375 try:
 Naming variables in catch clauses
376 try:
 Naming variables of catch clauses
377 try:
 Unhandled exception
378 try:
 Try with multiple catch clauses
379 try:
 try-finally without catch
400 functors:
 Check prepare is executed when functor is applied
401 functors:
 Check prepare declarations are availeble in define code
402 functors:
 Location of import + use imported functions
403 functors:
 Default esport name
404 functors:
 Named functor
405 functors:
 Functor without prepare
406 functors:
 Functor without import
407 functors:
 Named functor with require
408 functors:
 Named functor with require
700 tofix:
 Pattern matching on procedure arguments when record is handled by Boot
 Record.makeDynamic
```

```
800 quicksort:
 Quicksort from the book CTMCP, lazy and eager versions
801 lazy pascal:
 Lazy computation of Pascal triangle
802 scie:
 Scie test
803 transactions:
 Transaction Manager
804 lift:
 Lift simulation. Added determinism with controlled delays and lifts called
805 tuplespace:
 Tuplespace and queues
806 bounded buffer:
 Bounded buffer, also using monitor
807 lisser:
 Lisser code
808 wrapper:
 Secure data storage wrapper
809 elaguer:
 Numbers list manipulation
810 exprcode:
 Pattern matchin, recursive calls, guards,...
811 minimal:
 Computes the K minimal elements of a list.  This program compares
 eager algorithms with a simple lazy algorithm.
812 server:
 Server with port object
813 josephus decl:
 Josephus problem with streams
814 lazymergesort:
 Lazy merge sort
815 barrier:
 Variations on barrier synchronization
816 logicgates:
 Logic gates simulations
817 memoisation:
 Declarative memoization in Oz
998 legacy:
 Legacy namer test
999 legacy:
 Legacy namer test
```

Appendix E

README

```
Introduction
------------
This is a compiler for the Oz language, written in Oz.
It has been developed for a master thesis.
It targets the Mozart2 virtual machine.

Prerequisite
------------
Using this compiler requires the Mozart2 virtual machine available
at http://sourceforge.net/projects/mozart-oz/

Directories:
------------
/src ------------------------------> all Oz source code
/tests/definitions ----------------> test definitions, i.e. oz code and expected output
/tests/results --------------------> output of last test run
/report/ --------------------------> Latex code of the report

Makefile:
---------
A Makefile is located in the root directory, giving access to these commands:

- make tests
  run all test suite

- make testsfrom from=$num
  run all tests with order number higher or equal to $num

- make test test=$num
  run test with order number $num

- make run
  compile and run Oz code in the file src/run.oz
  Gives complete debug output
```

```
- make desc
  prints descriptions of all defined tests

- make clean
  deletes all compiled files

- make report
  generates the PDF report from Latex sources. Result file is report/report.pdf

- make compile src=path/to_file.oz
  compiles the file path/to_file.oz to path/to_file.ozf
```

Appendix F

Performance measures

Here are the results of performance tests. Only averages and standard deviations are included here. Exhaustive results are available in the file `perfs/timings.ods`.

The code tested on the new virtual machine is located in the file `perfs/timing.oz`, and the code tested on the Mozart1 virtual machine is in the file `perfs/timing.oz`.

Here is the procedure to reproduce the performance tests. First, uncomment the function you want to run and its call. Unused functions are commented to speed up the compilation. Then, issue the following command from the root directory of the project to get the measures of the new compiler:

```
make compile src=perfs/timing.oz && for i in $(seq 20); do
                                        ozengine perfs/timing.ozf; done
```

For measure of the Mozart1 compiler on the Mozart2 virtual machine:

```
ozc -c timing.oz && for i in $(seq 20); do ozengine timing.ozf  ; done
```

For measure on the Mozart1 platform:

```
ozc -c timing1.oz && for i in $(seq 20); do
                                     ozengine timing1.ozf  ; done
```

The column "New Compiler" lists timings of the code generated by the compiler described in this document, running on the Mozart2 virtual machine. The column "Old Compiler" holds the same measurement for code generated by the Mozart1 compiler targeting the Mozart2 virtual machine. The column "Mozart1" lists values for the same code compiled and run on Mozart1. Columns are arranged from left to right from the slowest to the fastest setup. The line "ratio" show how many times slower the setup of one column is compared to the setup of the column at its right.

Each of the following sections list the code of the function called, the argument passed, and the table of average and standard deviation of times measured.

F.1 Additions

The following code was called with `N=100000`, yielding results in the table. The body of the loop is simply 100 addition operations, as displayed in Figure F.1.

```
proc {Addition N}
   for I in 1..N do
      _ = 1+1 % 100 times
   end
end
```

Figure F.1: Addition

Additions			
	New Compiler	Old Compiler	Mozart1
average	2775.75	610.7	59.5
standard deviation	25.48	6.46	3.94
ratio	4.55	10.26	

F.2 Cells

The first test done was combining read and writes. The function `CellAccess` listed in Figure F.2 was called with `N=100000`.

```
proc {CellAccess N}
   C={NewCell 0}
in
   for I in 1..N do
      C:=@C+I  %100 times
   end
end
```

Figure F.2: Cell operations

Cell reads and writes			
	New Compiler	Old Compiler	Mozart1
average	6267.1	1471.8	1480.5
standard deviation	47.14	10.85	21.64
ratio	4.26	0.99	

Reads were also tested separately. The function `CellRead` in Figure F.3 was called with `N=100000`.

```
proc {CellRead N}
   C={NewCell 0}
in
   for I in 1..N do
      _=@C   % 100 times
   end
end
```

Figure F.3: Cell reads

Cell reads			
	New Compiler	Old Compiler	Mozart1
average	2500.25	399.05	104
standard deviation	22.18	5.45	5.03
ratio	6.27	3.84	

Finally, write operation were also tested separately. The function `CellWrite` in Figure F.4 was called with `N=100000`.

```
proc {CellWrite N}
   C={NewCell 0}
in
   for I in 1..N do
      C:=1
   end
end
```

Figure F.4: Cell reads

Cell writes			
	New Compiler	Old Compiler	Mozart1
average	1402.5		466.65 174
standard deviation	16.55	11.33	5.03
ratio	3.01	2.68	

F.3 Fibonacci

A naive Fibonacci function, `SlowFib` in Figure F.5, was used to compute the 30th Fibonacci number.

```
fun{SlowFib N}
   if N==0 then 0
   elseif N==1 then 1
   else
      {SlowFib N-2}+{SlowFib N-1}
   end
end
```

Figure F.5: A non-optimised Fibonacci function

30th Fibonacci			
	New Compiler	Old Compiler	Mozart1
average	2671.7		1029.15 183
standard deviation	7.63	10.82	4.70
ratio	2.60	5.62	

F.4 Pattern Matching

Pattern matching was tested with the function `OpenMatch` in Figure F.6 taking two arguments:

- The value to test against the case clauses. This was done to prevent the Mozart1 compiler's static analysis to completely transform the case instruction. Two values were tested: the constant `2` and the record `rec(d:1 z:3 y:4 x:6)`.
- The number of times the `case` instruction has to be performed. In our case, `N=1000000`.

The body of `OpenMatch` has a structure advantageous to the new virtual machine, which can combine all clauses in one instruction, whereas the Mozart1 virtual machine requires one instruction per clause.

```
proc {OpenMatch A N}
   for I in 1..N do
      case A
      of 1 then
         skip
      [] rec(a:_ ...) then
         skip
      [] rec(b:_ ...) then
         skip
      [] rec(c:_ ...) then
         skip
      [] rec(d:_ ...) then
         skip
      [] rec(e:_ ...) then
         skip
      [] rec(f:_ ...) then
         skip
      [] 2 then
         skip
      end
   end
end
```

Figure F.6: Matching open record patterns

30th Fibonacci			
	New Compiler	Old Compiler	Mozart1
average	12878.65		10384.25 2045
standard deviation	114.83	114.73	20.39
ratio	1.24	5.08	

Appendix G

Structure of code naming classes

```
% Name class, then switch to a new environment when inside the class
NamedClass={NamerForBody Var Params}
{NewParams.env backup()}

% Name method first, so the private methods are available in the
%methods bodies
NamedLabelsMethods={List.map Methods
                            fun {$ I}
                               {NameMethodLabel I NewParams}
                            end }
% Name methods arguments and bodies. Remember each method creates
% its own environment
NewMethods={List.map NamedLabelsMethods
                     fun {$ I}
                        {NameMethod I NewParams}
                     end }
% Name attributes, features,..
NewSpecs={List.map Specs fun {$ I} {NamerForBody I NewParams} end }

% Add initialisation code if needed
if @(NewParams.init)\=nil then
   ClassWithInit={WrapInFAnd
                     {List.append
                        [fClass(NamedClass NewSpecs NewMethods Pos)]
                        @(NewParams.init)}}
else
   ClassWithInit=fClass( NamedClass NewSpecs NewMethods Pos)
end

% Add declaration code if needed
if @(NewParams.decls)\=nil then
   ClassWithDecls=fLocal({WrapInFAnd @(NewParams.decls)}
                         ClassWithInit
                         Pos)
else
   ClassWithDecls=ClassWithInit
end

% Restore previous environment
{NewParams.env restore()}

% Return new AST for class
ClassWithDecls
```